Software Design Description for

Electronic Commerce Processing Node

Version 2.2

June 1999

Inter-National Research Institute, Inc.
12350 Jefferson Avenue, Suite 400
Newport News, Virginia 23602

Software Design Description

Contents

List of Appendices

List of Figures

List of Tables

This page has been intentionally left blank.

1.0 Scope

This Software Design Description (SDD) applies to Version 2.2 of the Electronic Commerce Processing Node (ECPN). This document follows the standards set forth in *Military Standard Software Development and Documentation* (MIL-STD-498) and in the associated Data Item Description (DID) for a Software Design Description (DI-IPSC-81435), as tailored by Inter-National Research Institute (INRI).

1.1 Identification

ECPN is a Computer System Configuration Item (CSCI) of the Electronic Commerce/ Electronic Data Interchange (EC/EDI) system.

1.2 System Overview

ECPN is being developed by INRI for the Defense Information Systems Agency (DISA). The role of ECPN is to serve as a single interface between the Government and its commercial trading partners for conducting EC/EDI. ECPN must ensure interoperability, economies of scale, and compliance to standards established by the Department of Defense (DoD) and Federal Program Office (PO).

The functional objectives of ECPN are to:

- Provide rigorous end-to-end accountability within the ECPN system, with no single point of failure that could result in loss or nondelivery of data
- Implement a Relational Database Management System (RDBMS) for storage of data passing through the ECPN
- Provide automated archive and retrieval mechanisms for messages and system configuration data
- Provide system performance information, including transaction statistics and communications status

1.3 Document Overview

This document describes the design for the ECPN CSCI of the EC/EDI system, including the allocation of requirements to the Computer Software Components (CSCs) and Sub-Computer Software Components (SCSCs) that compose ECPN.

This document contains the following sections and appendices:

Scope
States the purpose of the EC/EDI system; describes the role of ECPN within EC/EDI; and states the purpose of this SDD. (Section 1.0)

Referenced Documents
Lists the documents applicable to this SDD. (Section 2.0)

CSCI-Wide Design Decisions
Addresses ECPN's behavioral design and the selection and design of the CSCs and SCSCs that make up this CSCI. (Section 3.0)

Architectural Design
Identifies the CSCs that compose ECPN and the concept of execution among these units. (Section 4.0)

ECPN CSCI Detailed Design
Describes the design decisions and any constraints associated with each CSC and SCSC of ECPN. (Section 5.0)

Requirements Traceability
Describes the traceability between the ECPN requirements in this SDD and the ECPN system requirements. (Section 6.0)

Acronyms
Defines the acronyms used in this SDD. (Section 7.0)

Alerts
Defines the ECPN alerts and the processes that generate each of them. (Appendix A)

System Capacities
Describes the capacities of the ECPN data repositories. (Appendix B)

Glossary
Defines many terms used throughout this SDD. (Appendix C)

Message Object Parse API
Describes the API that allows you to specify a segment filter and apply it to a message object. (Appendix D)

2.0 Referenced Documents

The following documents are referenced in this SDD. In the event of a later version of a referenced document being issued, the later version shall supersede the referenced version.

- *Cleo 3780Plus User's Guide,* Interface Systems, Inc., May 1995.
- *Data Item Description - Software Design Description* (DI-IPSC-81435), December 1994.
- *Electronic Data Interchange Draft Version 3 Release 5 X12 Standards,* Data Interchange Standards Association, Inc., December 1994.
- *Federal Acquisition Guidelines (FAR), Draft Federal Government Implementation Guidelines,* Part 10, July 1997.
- *Kermit: Specification and Verification,* Huggins, James K., EECS Department, University of Michigan, Ann Arbor, MI.
- *Mercator: Execution Engine Core API Reference Guide*, TSI International Software, Ltd., 1997.
- *Mercator: Map Editor Reference Guide,* TSI International Software, Ltd., 1997.
- *Military Standard Software Development and Documentation* (MIL-STD-498), Department of Defense, December 1994.
- *Software Requirements Specification for Electronic Commerce Processing Node,* Version 2.2, April 1999.

3.0 CSCI-Wide Design Decisions

The selection and design of the CSCs that make up this CSCI are based on Section 3.0 of the *Software Requirements Specification (SRS) for Electronic Commerce Processing Node.*

4.0 Architectural Design

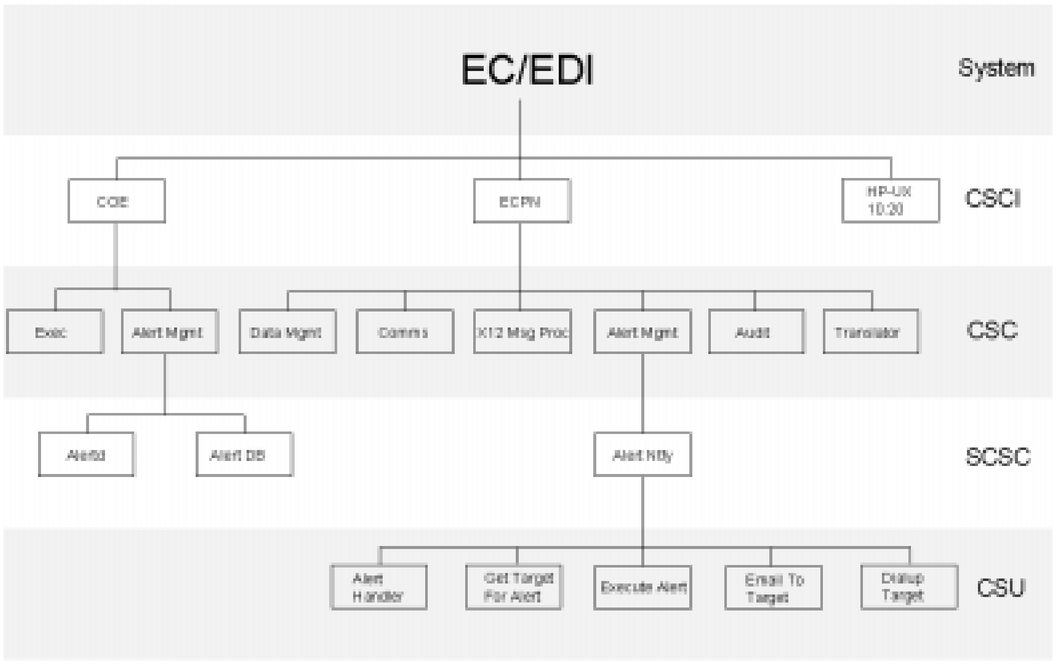
This section describes the following architectural design elements of ECPN:

- Architectural overview
- Computer Software Components (CSCs)
- Message processing flow
- Interface design

4.1 Architectural Overview

Figure 4-1 depicts the architectural layout of the system. Note that SCSCs and Computer Software Units (CSUs) are shown only for the Alert Management CSC, but are representative of the SCSCs and CSUs for each CSC. (For definitions of these terms, see Appendix C.)

Figure 4-1 ECPN Component Decomposition



4.2 Computer Software Components

This section identifies the ECPN CSCs, describes their high-level functions, and summarizes their constituent SCSCs.

4.2.1 Data Management

The Data Management CSC provides the data access model and software units for managing databases, logs, queues, and message information. The Data Management CSC provides multi-user and distributed access to the information it maintains. This CSC consists of the RPCServer, the message object SCSCs, and the system setup database.

4.2.2 Communications

The Communications CSC is responsible for transmitting and receiving UDF and X12 messages using various protocols; managing the communication channels; and reporting channel status.

The Communications CSC supports asynchronous serial communication with the Kermit® and ZMODEM protocols; bisynchronous serial communication with the CLEO® protocol; and network-based Transmission Control Protocol/Internet Protocol (TCP/IP) communication via FTP and electronic mail (Simple Mail Transport Protocol [SMTP] with Multi-purpose Internet Mail Extension [MIME]). This CSC consists of: EditChannels, Comms (Kermit, CLEO, ZMODEM, FtpComms), Ftpd, email-meta/emailsend, emaild, ChanStat, incoming and outgoing channel queues, and the channel database.

4.2.3 X12 Message Processing

The X12 Message Processing CSC contains those SCSCs and data elements used to interpret X12 messages and route messages to the communication channels. This CSC also contains the graphical user interface (GUI) applications that manage the routing of messages. The primary component of the X12 Message Processing CSC is the Router.

4.2.4 Translation

The Translation CSC converts UDF messages to X12 messages and also converts X12 messages to UDF messages. The primary components of the Translation CSC are the InXlator, OutXlator, and TPProfile (trading partner database).

4.2.5 Audit

The Audit CSC is responsible for creating and managing an audit trail for all messages processed by ECPN. The primary components of the Audit CSC are a RDBMS, message log, error queue, channel logs, channel queues, and email domain queue.

4.2.6 Alert Management

The Alert Management CSC provides a single mechanism for generating and managing alerts across the ECPN CSCI. The components of this CSC are the alert daemon, alert notifier, alert notifier database, and alert database.

4.2.7 Executive

The ECPN CSCI uses the COE CSCI Executive for launching and managing the processes of ECPN.

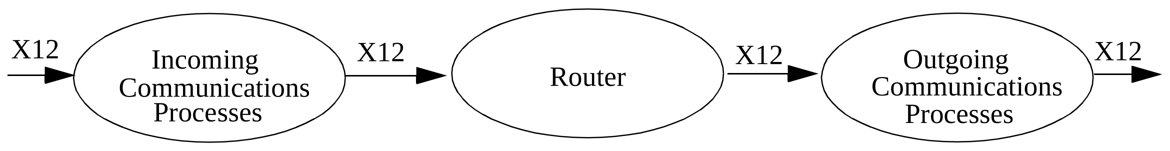
4.3 Message Processing Flow

This section describes the high-level message processing flow of the ECPN CSCI. The system processes two categories of messages–X12 and UDF. X12 is the ANSI benchmark for EC/EDI. UDF is a general term meaning “user-defined file”. Each system that interfaces with ECPN requires a different UDF. ECPN converts UDFs to X12 format (as described in Section 5.5) for processing and, if necessary, converts them back to UDF format for delivery.

4.3.1 X12 Message Processing

When an X12 message is received by incoming communications, it is passed to the Router. The Router processes the message, determines the intended recipients, and identifies the appropriate outgoing communications channel(s). If the message is to be sent as an X12, it is forwarded directly to the outgoing communications channel(s).

Figure 4-2 X12 Message Processing


X12
Incoming Communications Processes
X12
Router
X12
Outgoing Communications Processes
X12

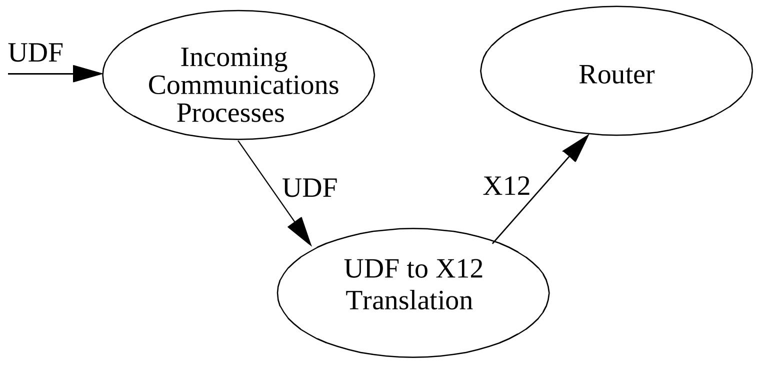
4.3.2 UDF Message Processing

UDF messages can be received and transmitted by the system. Because the primary message type processed by EPCN is X12, the system must translate all UDFs to X12 in order to archive and route the messages in a consistent manner.

4.3.2.1 Incoming UDF

Incoming UDF messages are queued from incoming communications to the Translator. The Translator then converts the messages from UDF to X12 and queues the converted X12 messages to the Router.

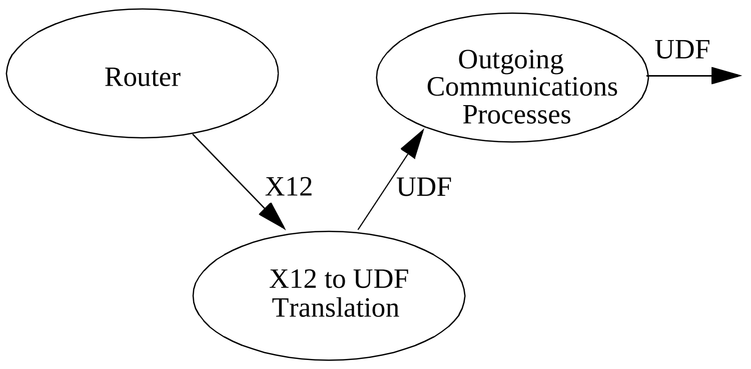
Figure 4-3 UDF Incoming Message Processing



4.3.2.2 Outgoing UDF

The Router processes the X12 message, determines the intended recipient, and identifies the appropriate outgoing communication channel(s). If the message is to be sent to a UDF-based channel, the Router queues the message to the X12 to UDF Translator for conversion. The resulting UDF message is then queued to outgoing communications and transmitted.

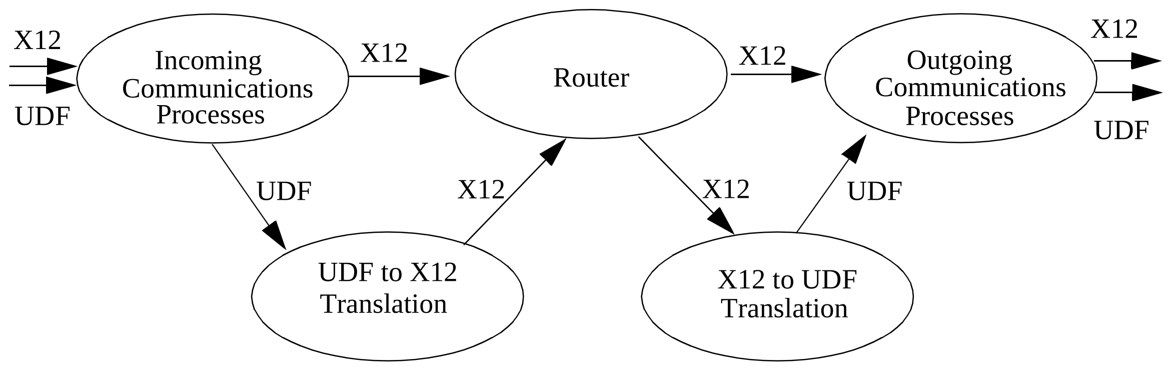
Figure 4-4 UDF Outgoing Message Processing



4.3.3 X12 and UDF Message Processing

Figure 4-5 depicts the entire message processing capability. Messages can be received and sent as either UDF or X12, but all messages processed by the Router must be in X12 format.

Figure 4-5 X12 and UDF Message Processing



4.4 Interface Design

ECPN is developed using open system components and standards. In its role as a communications relay between government and industry, ECPN relies upon commercially available communication interfaces. When possible, ECPN builds upon existing commercial off-the-shelf (COTS) software and adheres to the de-facto standards relevant to a given communications protocol. The communication standards and COTS products used by ECPN are outlined in Table 4-1.

Table 4-1 Communication Interface COTS Products and Standards

Communication Type	COTS Product	Standards
FTP	FTP libs 4.0[1], Wu-ftpd 2.4.2-B12[1]	RFC 793, RFC 959
Email (SMTP and MIME)	c-client, Sendmail 8.7.6	RFC 793, RFC 821 (SMTP), RFC 2045 (MIME)
Kermit	Kermit 6.0	Kermit Specification and Verification
ZMODEM	rz 3.42, sz 3.40	N/A
CLEO	CLEO 05265 (3780)	Cleo 3780Plus User's Guide

[1] Modified by INRI.

For more detailed information, see the *Interface Design Description (IDD) for the Electronic Commerce Processing Node.*

5.0 ECPN CSCI Detailed Design

This section describes the ECPN CSCs, their constituent SCSCs, and their primary data elements. The CSC descriptions include a summary of the SCSCs that comprise each CSC along with any constraints, limitations, or unusual features in the design of the software unit. Unless otherwise noted, all CSUs are written in the C programming language.

5.1 Data Management

The Data Management CSC provides access and management interfaces for the various databases, queues, logs, and message storage mechanisms used within EPCN. The CSUs that make up the Data Management CSC are stored within the Data Management local and remote libraries. The CSUs within these libraries implement the same application programming interface (API), but the data access method used by each library varies: the local library routines use file input / output (I / O), and the remote library components use remote procedure calls (RPCs). The local library is used by core processes that require high-speed access to data (e.g., the Router). The remote library is used by GUI applications and provides access to data on a remote computer system.

There are two distinct views, families of APIs, and methods for accessing data: local and remote. Local access to data is performed using file I/O, while data integrity is ensured by using file and record locking to serialize data access between processes. Data accessed via the local versions of an API is always consistent and up-to-date. Local access is used by core ECPN processes such as the Router and provides high-performance data access. Local data is accessed via the APIs stored in the local library.

Conversely, an application requiring remote data access uses RPCs to obtain the same data from the RPCServer. Once returned from the RPCServer, the data records are stored in dynamically allocated memory, representing a snapshot of the database. Because the remote library maintains a snapshot of the data, that data may become out-of-date with the data in the file (whether a remote or local file). Database and record versioning are used to ensure that operations on out-of-date records do not occur. Remote data access is supported by the APIs stored in the remote library.

To illustrate remote access, consider two users viewing data from record 1 in a database. While user A is viewing the data, user B is also viewing the same data, modifies part or all of the record, and stores the modified record in the database. At this point, user A is looking at an out-of-date version of record 1. When user A modifies and then attempts to store data, the remote library code silently passes version information about the record to the RPCServer. By comparing the current version of record 1 with the version passed in the transaction, the RPCServer determines that user A’s modification request is based on an old version of record 1 and returns an OUT-OF-DATE error to the application. If requested, user A’s application can issue a db_update() call to synchronize the remote snapshot of the data with the current version stored on disk.

Another distinction between local and remote data access is that the remote versions of the APIs cannot lock resources.

The Data Management CSC consists of the following SCSCs:

- RPCServer
- Databases
- Logs
- Queues
- Hash Tables
- Message Object

Within the context of the Data Management CSC description, the following definitions apply:

Database
A database provides random access to the records it contains. Records within the database are unordered and may be retrieved by record number or by starting at the first record and iterating to each subsequent record.

Log
Records within a log cannot be deleted and may only be added by appending the record to the log.

Queue
Queues maintain stores of records grouped by precedence. Individual records are retrieved by specifying the precedence of the record to be retrieved. Records can also be deleted.

Hash Table
Hash tables provide very fast lookups of records based on fields within the record.

5.1.1 RPCServer

The RPCServer manages requests from applications requiring remote access to data. The RPCServer provides access to message objects (as described in Section 5.1.6), databases, logs, and queues.

5.1.1.1 Database, Log, and Queue Rectifications

Database rectifications are used to perform server-side actions on a database, log, or queue whenever a database entry is modified, added, or deleted. Rectification routines perform checks to determine whether a requested action should be allowed. They also ensure that synchronization occurs between multiple RPC databases. For example, a rectification routine is used to ensure that each time a channel is added to the channels database, a corresponding entry is added to the channel status database.

To install a rectification routine, do the following:

a. Add the rectification routine to db_rect.c (in src/c/system/RPCServer). The arguments to every rectification routine are the same, and are as follows:

   – **trans**: The action being performed (DBT_STORE, DBT_MERGE, or DBT_DELETE)

   – **rec_num**: The database record number being acted on

   – **db_ptr**: A pointer to the record on disk being acted on (NULL if the RPCServer is performing an append). This argument is of the same type as the entries in the database.

   – **item**: A pointer to the record being changed or added (NULL if the RPCServer is performing a delete). This argument is of the same type as the entries in the database.

   – **filename**: A pointer to the database filename (currently unused)

   – **username**: The user requesting the operation (for future security checks)

   – **host**: The host from which the operation is being requested (for future security checks)

   – **open_db**: A pointer to the (already open) database, log, or queue.

b. Add a function declaration for the rectification routine to src/c/inc/EC/db_rect.h.

c. Add the rectification function to the db_tab[] table. The db_tab[] table is defined in src/c/inc/EC/db_tab.h. The rectification routine is the third argument in the table entry.

Each time the RPCServer performs a store or delete operation, the rectification routine (if it exists) is run, and the return value is checked before continuing with the operation. The database APPEND operation may result in two calls to the rectification routine. First, if an existing record is being replaced, a DELETE operation calls the rectification routine. Second, the database rectification routine is called from within the STORE operation. To differentiate between STORE and APPEND operations, check the db_ptr argument—it will be NULL for APPENDs.

5.1.1.2 Database, Log, and Queue Creation

Databases, logs, and queues are created by processes using the appropriate open call, e.g., db_open() or log_open(), with the DB_CREATE flag set.

5.1.2 Databases

A database provides random access to the records it contains. Records within the database are unordered and may be retrieved by record number or by starting at the first record and iterating to each subsequent record.

5.1.2.1 Database API

The following functions make up the database API. Most of these functions operate on the RPC_DATABASE handle that is obtained when the database is opened with the db_open() function.

int
db_alloc(RPC_DATABASE *dbp)
db_alloc() returns the record number of an available record in the database or DB_EOF if no free records exist.

int
db_append(RPC_DATABASE *dbp, const void *data)
db_append() behaves like db_store() except that the data is stored in the first free record. db_append() returns the record number where the data was stored on success or DB_EOF on error.

unsigned long
db_capacity(RPC_DATABASE *dbp)
db_capacity() returns the capacity that was used when the database was opened.

void
db_close(RPC_DATABASE *dbp)
db_close() closes the database and frees all associated memory.

int
db_delete(RPC_DATABASE *dbp, int rec_no)
db_delete() deletes the record ‘rec_no’. This record will be marked free, its data will be erased, and it will not be available for fetching. db_delete() returns 1 on success and 0 on error.

const char *
db_error(RPC_DATABASE *dbp)
db_error() returns a null-terminated string describing the last error associated with the database represented by dbp. Note that db_error() and the value of dbp->err are valid only when a database action fails. Their behavior is undefined if they are used after a successful operation.

int
db_fetch(RPC_DATABASE *dbp, int rec_no, const void *data)
db_fetch() fetches the data associated with record number ‘rec_no’ and places that data into the area pointed to by ‘data’. db_fetch() returns the record number on success or DB_EOF if an error occurs. Records that have not already been stored (i.e., free records) may not be fetched.

char *
db_filename(RPC_DATABASE *dbp)
db_filename() returns a null-terminated string representing the filename associated with database ‘dbp’.

int
db_first(RPC_DATABASE *dbp)
db_first() returns the record number of the first stored record in the database.

int
db_free(RPC_DATABASE *dbp)
db_free() returns the record number of the first free record in the database.

DB_REC_HDR *
db_hdr(RPC_DATABASE *dbp, int record)
db_hdr() returns a pointer to the database record header represented by record. This function should not normally be used by most applications.

unsigned long
db_id_num(RPC_DATABASE *dbp)
db_id_num() returns the ‘id’ of database ‘dbp’. This ‘id’ is used to determine whether the database needs to be converted.

unsigned long
db_in_use(RPC_DATABASE *dbp)
db_in_use() returns the number of stored records in the database.

int
db_last(RPC_DATABASE *dbp)
db_last() returns the record number of the last stored record in the database.

void
db_lock(RPC_DATABASE *dbp)
db_lock() will lock an entire database. As in the use of file locks, db_lock() will only lock out those processes that attempt to lock the database before use. This operation is a no-op for applications that access data remotely, because such applications cannot lock data.

int
db_lock_record(RPC_DATABASE *dbp, int rec_no, int block)
db_lock_record() locks the record ‘rec_no’. As in the use of file locks, db_lock_record() will only lock out those processes that attempt to lock the same record before use. This function will block the caller if the block parameter is non-zero. This operation is a no-op for applications that access data remotely, because such applications cannot lock data.

int
db_lookup(RPC_DATABASE *dbp, void *keyp, int nbytes)
db_lookup() performs a lookup of the key ‘keyp’ in the associated database hash table, assuming one is available. If a hash table is not available or the key is not found, DB_EOF is returned. If the hash table is available and the key is found, the record number where the key may be found is returned. For a complete description of the RPC database hash table, see Section 5.1.5.2.

long
db_ltom(RPC_DATABASE *dbp)
db_ltom() returns the last time of modification for database ‘dbp’.

char *
db_mod_host(RPC_DATABASE *dbp)
db_mod_host() returns the hostname associated with the last modification of database ‘dbp’.

int
db_mod_uid(RPC_DATABASE *dbp)
db_mod_uid() returns the user id of the last user that modified database ‘dbp’.

int
db_next(RPC_DATABASE *dbp, int rec_no)
db_next() returns the record number of the record following ‘rec_no’. If the record ‘rec_no’ is free, db_next() will return the next free record. If the record ‘rec_no’ is not free, db_next() will return the next ‘stored’ record. db_next() will return DB_EOF on error or when it reaches the end of the database. The following line of code is commonly used to run through the ‘stored’ records in a database:

```
for (r=db_first(dbp); r != DB_EOF; r=db_next(dbp, r))
   ;
```

RPC_DATABASE *
db_open(const char *filename, int mode, DBF_FORMATS format, unsigned long capacity)
db_open() will open a database with the given filename. The ‘format’ parameter is one of the enumerated types DBF_FORMATS found in db.h and specifies the data that is stored in the database. This format is mapped to a record type in db.h and db_tab.c. The capacity is the maximum number of entries allowed in the database. For ‘unlimited’ capacity, set this value to MAXINT. This function returns an RPC_DATABASE * that will be used in all the other database APIs or NULL on error.

int
db_prev(RPC_DATABASE *dbp, int rec_no)
db_prev() works the same as db_next() except it returns the previous record number, instead of the next record number. db_prev() returns DB_EOF when it reaches the beginning of the database.

int
db_rec_free(RPC_DATABASE *dbp, int record)
db_rec_free() returns a non-zero value if the database record ‘record’ is free. Otherwise, it returns 0.

unsigned long
db_rec_size(RPC_DATABASE *dbp)
db_rec_size() returns the size (in bytes) of a record stored in database ‘dbp’.

unsigned long
db_rec_version(RPC_DATABASE *dbp, int record)
db_rec_version() returns the database version for record ‘record’.

unsigned long *
db_records(RPC_DATABASE *dbp)
db_records() returns a pointer to an array containing the record number of all records stored in the database. This array of record numbers is generated by the db_first()/db_next() sequence above. The db_records call is useful when an application needs to fetch the xth through the yth element in a database.

void
db_set_id_num(RPC_DATABASE *dbp, unsigned long id_num)
db_set_id_num() sets the database ‘id’ for database ‘dbp’. This ‘id’ will later be used to determine whether a database needs to be converted.

int
db_store(RPC_DATABASE *dbp, int rec_no, const void *data)
db_store() stores the data represented by the ‘data’ parameter in the record ‘rec_no’. If that record was a free record, it will automatically be marked ‘in_use’ and will be available for fetching. db_store() returns 1 on success or 0 on error.

void
db_sync(RPC_DATABASE *dbp)
db_sync() writes all memory-mapped data for a given database to disk. This operation is not normally necessary for applications.

void
db_unlock(RPC_DATABASE *dbp)
db_unlock() releases the lock on a database obtained via db_lock(). This operation is a no-op for applications that access data remotely, because such applications cannot lock data.

void
db_unlock_record(RPC_DATABASE *dbp, int rec_no)
db_unlock_record() will release the lock on record ‘rec_no’ obtained via db_lock_record(). This operation is a no-op for applications that access data remotely, because such applications cannot lock data.

int
db_update(RPC_DATABASE *dbp)
db_update() updates the data in the remote database copy to match the copy on the server. This operation is a no-op for applications that access data locally, because such applications use direct file I/C.

void
db_update_ltom(RPC_DATABASE *dbp, int uid, const char *host)
db_update_ltom() updates the last time of modification (to the current time), the user (uid) that last modified the database 'dbp', and the hostname from where the last modification was originated.

unsigned long
db_version(RPC_DATABASE *dbp)
db_version() returns the global database version number.

void
disable_db_cache(void)
disable_db_cache() will disable the database cache. Internally, the database code caches the most recently used databases. When an application calls db_close(), the database data is flushed to disk, but the database is not necessarily unmapped. This allows for the efficient opening of frequently used databases. Disabling the database cache will seriously impact application performance in most cases, so it should be used with caution.

void
enable_db_cache(void)
enable_db_cache() will enable the database cache after it has been disabled by calling disable_db_cache(). The database cache is enabled by default, so this function will only be necessary if disable_db_cache() has been used. For a complete description of the database cached, see disable_db_cache().

5.1.3 Logs

The log API provides methods to create and manipulate a log. Allowable API operations include appending to the end of the log and modifying the data in a record. Deleting entries is not allowed.

5.1.3.1 Log API

int
log_append(RPC_LOG *logp, const void *data)
log_append() behaves like log_store() except that the data is stored in the first free record. log_append() returns the record number where the data was stored on success or DB_EOF on error.

unsigned long
log_capacity(RPC_LOG *logp)
log_capacity() returns the capacity that was used when the log was opened.

void
log_close(RPC_LOG *logp)
log_close() will close ‘logp’ that was opened via a log_open () call and free all associated memory.

int
log_fetch(RPC_LOG *logp, int rec_no, const void *data)
log_fetch() will fetch the data associated with record number ‘rec_no’ and place that data into the area pointed to by ‘data’. log_fetch() returns the record number on success or DB_EOF if an error occurs. Records that have not already been stored (i.e., free records) may not be fetched.

int
log_free(RPC_LOG *logp)
log_free() returns the record number of the first free record in the log.

int
log_first(RPC_LOG *logp)
log_first() returns the record number of the first stored record in the log.

unsigned log
log_in_use(RPC_LOG *logp)
log_in_use() returns the number of stored records in the log.

int
log_last(RPC_LOG *logp)
log_last() returns the record number of the last stored record in the log.

void
log_lock(RPC_LOG *logp)
log_lock() will lock an entire log. As in the use of file locks, log_lock() will only lock out those processes that attempt to lock the log before use. This operation is a no-op for applications that access data remotely, because such applications cannot lock data.

int
log_next(RPC_LOG *logp, int rec_no)
log_next() returns the record number of the record following ‘rec_no’. If the record ‘rec_no’ is free, log_next() will return the next free record. If the record ‘rec_no’ is not free, log_next() will return the next ‘stored’ record. log_next() will return DB_EOF on error or when it reaches the end of the log. The following line of code is commonly used to run through the ‘stored’ records in a log:

```
for (r=log_first(logp); r != DB_EOF; r=log_next(logp, r))
  ;
```

RPC_LOG *
log_open(const char *filename, int mode, DBF_FORMATS format, unsigned long capacity)
log_open() will create a database with the given filename. The ‘format’ parameter is one of the enumerated types DBF_FORMATS found in db.h and specifies the data that is stored in the database. This format is mapped to a record type in db.h and db_tab.c. The capacity is the maximum number of entries allowed in the database. For unlimited capacity, set this value to MAXINT. This function returns an RPC_LOG * that will be used in all the other log APIs or NULL on error.

int
log_prev(RPC_LOG *logp, int rec_no)
log_prev() works the same as log_next() except it returns the previous record number, instead of the next record number. log_prev() returns DB_EOF when it reaches the beginning of the log.

unsigned long *
log_records(RPC_LOG *logp)
log_records() returns a pointer to an array of records. This array of records is generated by the log_first()/log_next() sequence above. The log_records call is useful when an application needs to fetch the xth through the yth element in a log.

int
log_store(RPC_LOG *logp, int rec_no, const void *data)
log_store() will store the data represented by the ‘data’ parameter in the record ‘rec_no’. If that record was a free record, it will automatically be marked ‘in_use’ and will be available for fetching. log_store() returns 1 on success or 0 on error.

void
log_unlock(RPC_LOG *logp)
log_unlock() will release the lock on a log. This operation is a no-op for applications that access data remotely, because such applications cannot lock data.

int
log_update(RPC_LOG *logp)
log_update() will update the data in the remote log copy to match the copy on the server. This operation is a no-op for applications that access data locally, because such applications use direct file I/C.

5.1.4 Queues

The queue API provides methods to create and manipulate multi-precedence, first-in first-out (FIFO) queues. The maximum number of precedences is 20.

Allowable API operations include appending to the end of the queue (at the correct precedence), modifying the data in a record, and deleting an entry from the queue. APIs also exist that allow a consumer of the queue data to block on an empty queue.

The queue API allows multiple consumers to process distinct entries within the same queue. Typically, this is done by having a master consumer process that monitors the growth of the queue and spawns additional consumer processes. The consumer processes act upon and remove entries in the queue until all entries have been processed, at which point all consumer processes, except the master consumer, terminate. The consumer processes use q_fetch_and_lock() to retrieve each record and, thereby, prevent another consumer process from retrieving the same record. Should a consumer process terminate abnormally, any locks placed on the record by that process using q_set_and_lock() will be removed. This action allows the record to be processed by another consumer process.

5.1.4.1 Queue API

int
q_append(RPC_QUEUE *qp, int prec, const void *data)
q_append() inserts the data represented by ‘data’ as the last element with precedence ‘prec’. q_append() returns the record number where the data was stored on success or DB_EOF on error. If the queue was empty before this append and the pid field in the queue header is non-zero, the calling process will send a SIGUSR1 signal to the process specified by ‘pid.’ For more detail see the description of q_fetch_and_lock().

void
q_close(RPC_QUEUE *qp)
q_close() takes an RPC_QUEUE * argument that was the result of a previous q_open() call. This function will close the queue and free all associated memory.

int
q_delete(RPC_QUEUE *qp, int rec_no)
q_delete() will delete the record ‘rec_no’. This record will be marked free, its data will be erased, and it will not be available for fetching. If the calling process holds a record lock on this record, q_delete() will release it. q_delete() returns 1 on success and 0 on error.

int
q_fetch(RPC_QUEUE *qp, int rec_no, const void *data)
q_fetch() will fetch the data associated with record number ‘rec_no’ and place that data into the area pointed to by ‘data’. q_fetch() returns the record number on success or DB_EOFon error. Records that have not already been stored (i.e., free records) may not be fetched.

int
q_fetch_and_lock(RPC_QUEUE *qp, void *data, struct timeval *tv)
q_fetch_and_lock() fetches and locks the next available record in the queue. Locking the individual record keeps that record from being used by other processes that use q_fetch_and_lock() to retrieve records. If more than one process is feeding from the same queue, they should all use q_fetch_and_lock(). While q_fetch() will fetch records, it will not obey the record locking that is implemented in q_fetch_and_lock(), thus allowing processes to step on each other. '*tv' represents the amount of time to block in this function while waiting for an entry. If 'tv' is NULL, q_fetch_and_lock() will block on an empty queue forever. If 'tv' is non-NULL, q_fetch_and_lock() will block for the time specified in 'tv'.

q_fetch_and_lock() installs a signal handler to handle the SIGUSR1 signal that will be sent by a process that appends to an empty queue.

q_fetch_and_lock() returns DB_EOF on a failed fetch or if a timeout occurs and the queue is still empty; otherwise, it will return the record number that was fetched and locked. q_fetch_and_lock() should not be used by those applications linked with the remote library.

int
q_first(RPC_QUEUE *qp)
q_first() returns the record number of the first stored record in the queue.

unsigned long
q_in_use(RPC_QUEUE *qp)
q_in_use() returns the number of stored records in the queue.

unsigned long
q_in_use_prec(RPC_QUEUE *qp, int prec)
q_in_use_prec() returns the number of stored records in the queue at the precedence 'prec'.

void
q_lock(RPC_QUEUE *qp)
q_lock() will lock an entire queue. As in the use of file locks, q_lock() will only lock out those processes that attempt to lock the queue before use. This operation is a no-op for applications that access data remotely, because such applications cannot lock data.

int
q_next(RPC_QUEUE *qp, int rec_no)
q_next() returns the record number of the record following 'rec_no' based on FIFO precedence ordering. db_next() will return DB_EOF on error or when it reaches the end of the queue. The following line of code is commonly used to run through the records in a queue in FIFO precedence order:

```
for (r=q_first(qp); r != DB_EOF; r=q_next(qp, r))
   ;
```

RPC_QUEUE *
q_open(const char *filename, int mode, QF_FORMATS format, unsigned long capacity)
q_open() will create a FIFO-precedence queue with the given filename. The ‘format’ parameter is one of the enumerated types QF_FORMATS found in db.h and specifies the data that is stored in the queue. This format is mapped to a record type in db.h and queue.c. The capacity is the maximum number of entries allowed in the queue. For unlimited capacity, set this value to MAXINT. This function returns either an RPC_QUEUE * to be used in all the other queue APIs or NULL on error.

int
q_prev(RPC_QUEUE *qp, int rec_no)
q_prev() works the same as q_next() except it returns the previous record number, instead of the next record number. q_prev() returns DB_EOF when it reaches the beginning of the queue.

int
q_rec_prec(RPC_QUEUE *qp, int rec_no)
q_rec_prec() returns the precedence of the record ‘rec_no’ or DB_EOF on error.

time_t
q_rec_toq(RPC_QUEUE *qp, int rec_no)
q_rec_toq() returns the time-of-queue for the record ‘rec_no’ or DB_EOF on error.

void
q_set_pid(RPC_QUEUE *qp, long pid)
q_set_pid() places the process id specified by ‘pid’ into the pid field of the queue. When a process appends an entry to an empty queue, the process will send a SIGUSR1 to the process identified by ‘pid’. Because the process identified by ‘pid’ has specified a signal handler for the SIGUSR1 signal, that process can be notified when an empty queue contains data and thereby avoid repeated polling to find non-empty queues. This operation is a no-op for applications that access data remotely, because such applications cannot lock data.

int
q_store(RPC_QUEUE *qp, int rec_no, const void *data)
q_store() will store the data represented by the ‘data’ parameter in the record ‘rec_no’. If that record was a free record, it will automatically be marked ‘in_use’ and will be available for fetching. q_store() returns 1 on success or 0 on error.

void
q_unlock(RPC_QUEUE *qp)
q_unlock() will release the lock on a queue. This operation is a no-op for applications that access data remotely, because such applications cannot lock data.

void
q_unlock_record(RPC_QUEUE *qp, int rec_no)
q_unlock_record() will release the lock on record ‘rec_no’. This operation is a no-op for applications that access data remotely, because such applications cannot lock data.

```
void
q_update(RPC_QUEUE *qp)
```

q_update() will update a remote application’s copy of the queue to match the copy on the server. This operation is a no-op for applications that access data locally, because such applications use direct file I/C.

5.1.5 Hash Tables

For those databases that require fast, efficient lookups, hash tables are available. The hash table code implements a disk-based hash table. The hash table API provides concurrent access between processes during lookups and is non-volatile and very tunable.

5.1.5.1 Hash Table Creation

The only code necessary to ensure that a hash table is created and maintained is the addition of ‘get_key()’ functions to the db_tab table (found in db_tab.h). These get_key() functions must match this prototype:

```
int
get_key(const void *datap, void **rsltp, int *nbytesp)
```

‘datap’ is a pointer to the data, which should be cast to the type of variable that is appropriate for this function. ‘*rsltp’ should point to the beginning of the field that should be entered in the hash table. *nbytesp should be filled in with the length (in bytes) of the key. An example of a get_key() function follows:

```
typedef struct {
char name[25];
int date;
} FOO;

int
get_name(void *datap, void **rsltp, int *nbytesp)
{
FOO *p = (FOO *)datap;
*rsltp = p->name;
*nbytes = strlen(p->name);
}
```

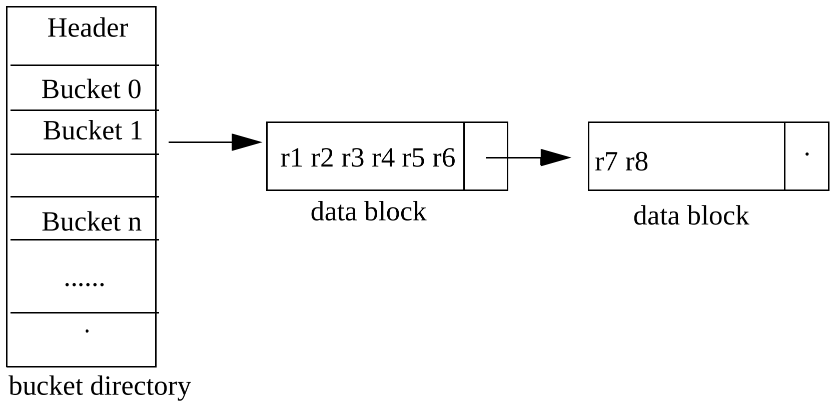
More than one get_key() function may be defined for a given database. The only restriction is that the actual keys that are added must be unique.

To make use of the hash table, use db_lookup() (described in Section 5.1.2.1).

5.1.5.2 Hash Table Implementation

A hash table consists of a bucket directory and a number of buckets. The bucket directory consists of one or more file blocks, and a bucket consists of exactly one block. The bucket directory is indexed by the hash value of the key. From that index into the bucket directory, the buckets are then searched for the key. The records in each bucket are sorted by the data value, so a binary search of the data is used to speed up the search. If the key is not found in the first bucket, any and all subsequent data blocks are searched until the last block is searched or the key is found.

Figure 5-1 Hash Table Data Structure



5.1.5.3 Hash Table API

When accessing a hash table that is associated with a database, the database API manages the hash table using the hash table API. This API should only be used directly when implementing hash tables *outside* of the scope of the RPC databases. The following is a description of the hash table API.

void
ht_close(HTBL *hp)
ht_close() closes a hash table and frees all associated memory.

int
ht_delete(HTBL *hp, void *keyp, int nbytes)
ht_delete() performs a lookup of ‘keyp’ and deletes the entry if found. ht_delete() will return 1 on success and 0 on failure.

unsigned long
ht_get_version(HTBL *hp)
ht_get_version() returns the version of a hash table.

int
ht_insert(HTBL *hp, int rec, void *keyp, int nbytes)
ht_insert() will insert ‘keyp’ into the hash table. ‘rec’ is also stored with ‘keyp’ and will be returned when a lookup of ‘keyp’ is performed. ht_insert() returns 0 on failure or the number of disk block accesses required to insert this entry on success.

int
ht_lookup(HTBL *hp, void *keyp, int nbytes)
ht_lookup() performs a lookup of ‘keyp’ and, if found, returns the associated ‘rec’. ht_lookup() returns -1 on error, or the ‘rec’ on success.

HTBL *
ht_open(char fname, int capacity, int maxkeysize, int blockspervalue)
ht_open() opens a disk-based hash table. ‘fname’ is the filename of the file to open. ‘capacity’ is the expected number of entries in the hash table. ‘maxkeysize’ is the number of bytes required by the largest key. ‘blockspervalue’ is the number of blocks the user wanted to be allocated to each hash value. ‘vers’ is the version of the hash table and it is user-defined.

ht_open() returns a NULL on error, or a pointer to a HTBL on success.

void
ht_print(HTBL *hp)
ht_print() prints information about a hash table, including each entry in the bucket directory and the total number of records in each entry.

void
ht_set_version(HTBL *hp, unsigned long vers)
ht_set_version() sets the version of a hash table.

5.1.5.4 Miscellaneous RPCServer Remote Procedure Calls

int
append_file(char *filename, char *buf, int buflen)
append_file() appends a buffer to an ASCII file.

int
rpc_fetch_file(char *fname, char **buffer, int size, char *hostname, char **error)
rpc_fetch_file() connects to the RPCServer on the specified host, and then fetches a file, filling the “buffer” variable with the contents of the file.

int
rpc_store_file(char *fname, char *buffer, int size, char *hostname, char **error)
rpc_store_file() connects to the RPCServer on the specified host, and then writes the contents of “buffer” to the specified file on the remote host.

int
rpc_append_file(char *fname, char *buffer, int size, char *hostname, char **error)
rpc_append_file() connects to the RPCServer on the specified host, and then appends the contents of “buffer” to the specified file on the remote host.
int rpc_msg_annotate(char *msn, char *annotation, char *hostname, char **error)
rpc_msg_annotate() annotates a message object through the RPCServer.

int
read_seg(char *filename, int offset, size_t size, char *buffer, char **error);
read_seg() reads a segment of “size” bytes from a file, beginning at “offset”, and stores the results in the variable “buffer”.

int
rpc_read_directory(char ***file_list, char *directory, int exclude_subdirs, char *hostname, char **error);
rpc_read_directory() reads the contents of a directory on a remote host through the RPCServer. If “exclude_subdirs” is set, only files are included in the resulting list.

int
rpc_delete_file(char *fname, char *hostname, char**error);
rpc_delete_file() connects to the RPCServer on a remote host and deletes the specified file.

5.1.6 Message Object

The message object is the primary storage place for raw and derived information on a single message. The message object contains the message text, any errors or alerts associated with the message, a list of actions taken on the message, as well as any ECPN administrator annotations. For the purpose of this description, a message is defined as all of the text within a single X12 ISA envelope (ISA/IEA pair). Each message object is stored on disk in an architecture-neutral, compressed data file, and the data files are stored in Daily/<yyyymmdd>/Archives/msg_objs. APIs exist for accessing each of the individual pieces of a message object (not shown here).

The message sequence number (MSN) uniquely identifies each message within the system. The format of the MSN is a one-letter site ID followed by an eight-digit counter, a slash, the four-digit year, the two-digit month, and the two-digit day on which the message was received (e.g., c00000001/19970721). The eight-digit counter is reset to 1 each day at midnight (Universal Time Coordinate [UTC]). This MSN format provides for storing a maximum of 99,999,999 ISAs each day.

Message objects are created by the Router process and updated by the outgoing X12 to UDF Translator and outgoing communications processes. User access to the message object content is available through several paths, including all applications that use the Journal Data Summary (JDS) Viewer (described in Section 5.4.7) and the Raw Viewer (described in Section 5.4.8).

5.1.6.1 Message Object API

The following functions are used to open, update, store, and close.

EC_MSG_OBJ *
open_msg_obj (char *MSN, int flags, int mode);
open_msg_obj() opens a message object (using the format nnnnnnnn/yyyymmdd). The flags and mode have the same meaning as in open(2). In addition, the message object is locked as shared mode, exclusive for read-only mode, or read-write opens mode, respectively.

int
store_msg_obj (char *MSN, EC_MSG_OBJ * m);
store_msg_obj() writes the given message object to disk, closes it, and frees it.

int
close_msg_obj (EC_MSG_OBJ * m);
close_msg_obj() closes a message object and frees it.

EC_MSG_OBJ *
alloc_msg_obj (void);
alloc_msg_obj() creates a new message object in memory only (not associated with a file). store_msg_obj() must be called to save that new message object to disk.

int
flush_msg_obj (EC_MSG_OBJ * m);
flush_msg_obj() writes an already open message object to disk. flush_msg_obj() simply calls write_msg_obj().

int
write_msg_obj (EC_MSG_OBJ * m, int fd);
write_msg_obj() writes a message object to disk.

int
detach_msg_obj (EC_MSG_OBJ * m);
detach_msg_obj() detaches a message object from its file, leaving only a copy in memory. If a message object is open read-write, it is flushed to disk before it is detached. This should be used by calling routines that only need a snapshot of the message object.

EC_MSG_OBJ *
reopen_msg_obj (EC_MSG_OBJ * m, char *new_name, int flags, int mode);
reopen_msg_obj() assigns a message object to a new file, and then closes the old file. This will work even if the old object is only memory-resident.

free_msg_obj(EC_MSG_OBJ *m)
free_msg_obj() frees up any memory associated with a message object.

5.1.6.2 Message Object Field Descriptions

Table 5-1 Message Object Fields

Field Name	Type	Description
version	int	Version number of the message object database
open_flags	int	open() flags--used internally
open_fd	int	File descriptor for message object--used internally
segments	flist_segs	Packages the following pieces (indicated by indention):
seg_type	int	Numeric hash value of segment type. Segment type is stored as an integer to provide fast segment lookups. The HASH macro converts a segment name into a hash value. This hash value is generated using a shift-left hash function on the segment ID string (e.g., an “ISA” segment ID would generate a seg_type value of ((‘I’ << 24) + (‘S’ << 16) + (‘A’ << 8) + ‘\0’)
seg_content	BINARYSTRING	Contains the entire X12 segment through the segment terminator (e.g., an ISA “line”)

Table 5-1 Message Object Fields (Continued)

Field Name	Type	Description
errors	flist_errs	Packages the following pieces (indicated by indention):
err_type	int	Numeric value depicting the type of error (e.g., ISA Parse Error). This value is derived from a CSC mask and a specific error value. The CSC masks are used throughout processing to identify the origin of an error. For a list of possible mask values, see Table 5-2. The specific error value is set but not currently used in processing.
msg_seg_num	int	Index of the segment in msg_seg[] in which the error was identified
msg_seg_offset	int	Offset of the segment in msg_seg[] in which the error was identified
error_expansion	wrapstring	Description string associated with the err_type. For a list of possible values, see Table 5-2.
chan_name	wrapstring	Outgoing channel where error occurred
x12_vals	x12_obj	Packages the following pieces (indicated by indention):
record	int	Record number of message object
source	int	Input channel mask
msgtype	wrapstring	Type of channel on which the message was originally received. Possible values: X12, DBMS 1.0, DIFMS, DWAS, GAFS, IFAS, IPC, ITIMP, LEGACY, SAACONS, SIFS, SPS, Stanfins 1.0, STARS.
msn	wrapstring	Message sequence number (format: nnnnnnnn/ yyyymmdd)
mask	int	Bitmask (0=int/ext, 1=user/non-user, 2=ascii/ binary data)
precedence	int	(Unused)
direction	int	VAN->GW or GW->VAN
logname	wrapstring	Relative path for message log
SrcChnl	wrapstring	Incoming channel name
InCharCount	int	Number of bytes in incoming source message
SrcChnlXref	wrapstring	XREF (3-letter identifier) associated with source channel

Table 5-1 Message Object Fields (Continued)

Field Name	Type	Description
ArchFilename	wrapstring	Filename of compression file that the incoming communications process used to store the original received content
RemoteInFileName	wrapstring	Incoming filename
IsaNum	int	ISA Control Number (ICN) (X12 ISA13)
ISATo	wrapstring	Receiver Interchange (X12 ISA08)
ISAFrom	wrapstring	Sender Interchange (X12 ISA06)
ElemSep	u_char	Element Separator (extracted from the 4th byte position of the incoming ISA segment)
SubElemSep	u_char	Subelement Separator (X12 ISA16)
SegTerm	u_char	Segment Terminator (extracted from the 106th byte position of the incoming ISA segment)
gs_list	flist_x12_gs	Packages the following pieces (indicated by indention):
flags	long	Bitmask indicating messages that were generated by the Translator (e.g., 824, 997)
ai_qual	wrapstring	ISA Line Authorization Information Qualifier (X12 ISA01)
ai	wrapstring	ISA Line Authorization Information (X12 ISA02)
si_qual	wrapstring	ISA Line Security Information Qualifier (X12 ISA03)
si	wrapstring	ISA Line Security Information (X12 ISA04)
send_qual	wrapstring	Sender Interchange Qualifier (X12 ISA05)
recv_qual	wrapstring	Receiver Interchange Qualifier (X12 ISA07)
i_dtg	int	Interchange DTG (integer representation of X12 ISA09 and X12 ISA10 fields)
ics_id	u_char	Interchange Control Standards ID (X12 ISA11)
ic_ver	wrapstring	Interchange Control Version Number (X12 ISA12)
ack	u_char	Acknowledgment Requested Indicator (X12 ISA14)
test	u_char	Test Indicator (X12 ISA15)
i_date_str	wrapstring	Interchange Date String (X12 ISA09)
i_time_str	wrapstring	Interchange Time String (X12 ISA10)

Table 5-1 Message Object Fields (Continued)

Field Name	Type	Description
InUDFFilename	wrapstring	Name of incoming UDF from which this message object was built
section	int	Section number for this X12 with respect to the group
tot_sections	int	Total number of other X12s in the group
section_keeper	string	Master X12 for the group
section_list	string []	List of other MSNs in the group
GS_index	int	Index of the entry in msg_seg[] that contains this GS segment
GE_index	int	Index of the entry in msg_seg that contains this GE segment
GrpCtrlNum	int	Group Control Number (X12 GS06)
GSTo	wrapstring	GS Receiver ID (X12 GS03)
GSFrom	wrapstring	GS Sender ID (X12 GS02)
fi_code	wrapstring	Functional Identifier Code (X12 GS01)
dtg	int	Date Time Group (DTG)
ra_code	wrapstring	Responsible Agency Code (X12 GS07)
vri_code	wrapstring	Version/Release/Industry ID (X12 GS08)
st_list	flist_x12_st	An st_list array is contained within each gs_list and contains information on one of the ST segments.
date_str	wrapstring	DTG Date String (X12 GS04)
time_str	wrapstring	DTG Time String (X12 GS05)
ST_index	int	Index of entry in msg_seg[] that contains this ST segment
SE_index	int	Index of entry in msg_seg[] that contains this SE segment
TransType	int	Numeric representation of the X12 transaction ID (ST01) (e.g., 850, 843, 836)
TransNum	wrapstring	Transaction Control Number (X12 ST02)
QualDtg	int	Date/time qualifier for BQR and BQT segments (X12 BQT03 and BQR03)
PurchaseOrder	wrapstring	Purchase Order Number for a BEG segment (X12 BEG03)

Table 5-1 Message Object Fields (Continued)

Field Name	Type	Description
solicitation	wrapstring	X12 Solicitation Number (X12 BQT02 and BQR02)
incoming_udf	UDF_vals	Contains information about incoming UDF to X12 translation
old_incoming_udf	flist_char	Original incoming UDF
error_type	int	Translation error (UDF_OK, TPDB_ERR, etc.)
xlate_rpt	wrapstring	Report data from translation
incoming_udf	BINARYSTRING	Incoming UDF (empty for outgoing)
linkages	flist_linkages	Links to other message objects
msn	wrapstring	MSN to which the message object is linked
type	u_char	Type of link (824, 997, etc.)
additional_info	wrapstring	Descriptive information about the link
oper_annotations	flist_annot	One entry exists for each annotation made to this message. Packages the following pieces (indicated by indention):
annot_text	wrapstring	Annotation text entered by the ECPN administrator
oper_name	wrapstring	Name of the ECPN administrator making the annotation
annot_time	long	Time that the annotation was made
host_name	wrapstring	Host from which the annotation was made
err_q_index	int	Index of this message object in error queue (-1 if not queued)
actions	flist_MSG_ ACTION	List of actions taken with this message object
type	ACTION_TYPE	Type of action taken. Possible values: AT_RECEIVE, AT_ROUTE, AT_PROCESS, AT_REXLATE, AT_REXMIT, AT_OPXMIT (future development), AT_EQ_DELETE, AT_MOD_TPDB, AT_X122UDF_XLATE, AT_X122UDF_XLATE, AT_SYSGEN, AT_REROUTE, AT_CHANQ_DEL, AT_UDF_ROUTE, AT_ROUTE_CANCEL, AT_ERROR, AT_OTHERS
time	long	Time the status was last set

Table 5-1 Message Object Fields (Continued)

Field Name	Type	Description
status	ACTION_STATUS	Status of action. Possible values: AS_COMPLETE, AS_FAILURE.
identifier	int	Numeric value that provides a cross-reference between the items in the actions list and items in the routes list. Not applicable for non-route/reroute actions.
str	wrapstring	Contains expansion information (e.g., ECPN administrator name/host, channel name, etc.)
str2	wrapstring	Contains additional expansion information
routes	flist_MSG_ROUTE	List of routes in the message object
chnl_name	wrapstring	Channel routed to
subaddr	flist_wrapstring	For a route to an email channel, this array of strings holds the complete list of addressees as taken from the "TO:" list in the email channel configuration. This field is not applicable for a non-email channel route.
identifier	int	A unique integer value is assigned for each route in a message object. This value is also used to build queue records for sending the message object to an outgoing communications process or the outgoing translator, thereby establishing a cross-reference.
reason	ROUTE_REASON	Reason for route (i.e., GS-TO). Valid values: RR_NOT_APP, RR_GS_FROM, RR_GS_TO, RR_ISA_FROM, RR_ISA_TO, RR_FILE_PATTERN_CASE, RR_FILE_PATTERN_NOCASE, RR_ALL_CHAN, RR_SYS_GEN, RR_RETRANSMIT, RR_MSG_REPORT, RR_MSG_REPORT_REXMIT, RR_SYS_GEN_ADMIN
gs_sts	flist_GS_ST	List of GS_ST records within the route. Each one represents a file to be transmitted.
old_time	long	Time that status was last set
gs	int	GS index (zero-based) of the gs_list portion of the message object. A value of -1 denotes all GSs in the message object.

Table 5-1 Message Object Fields (Continued)

Field Name	Type	Description
st	int	ST index (zero-based) of the st_list within the above indexed gs_list record. A value of -1 denotes all STs within the gs_list record.
old_subaddr_bm	bitmask	Bitmask denoting which indices in the route record subaddressee list are targeted in this particular route portion
old_filename	wrapstring	Name (or for email, the message ID) of the transmitted file
old_status	ROUTE_STATUS	Status of route (transmitted/pending). Valid values: RT_QUEUED, RT_XMITTING, RT_COMPLETE, RT_FAILURE, RT_CANCELLED, RT_OPDEL
old_transmitter_pid	int	PID of the communication process handling the transmission
old_ack_997	msn_link	Link to the 997 generated from this route
msn	wrapstring	Message Sequence Number
type	u_char	Type of ack (e.g., 997, 824)
additional_info	wrapstring	Descriptive information
route_type	ROUTE_TYPE	Type of route. Valid values: RTYPE_NORMAL, RTYPE_ACK997, RTYPE_CC
msn_linkages	flist_linkages	Links to other message objects
msn	wrapstring	MSN to which the message object is linked
type	u_char	Type of link (e.g., 824, 997)
additional_info	wrapstring	Descriptive information
time	long	Time that status was last sent
filename	wrapstring	Name (or for email, the message ID) of the transmitted file
status	ROUTE_STATUS	Status of route (transmitted/pending). Valid values: RT_QUEUED, RT_XMITTING, RT_COMPLETE, RT_FAILURE, RT_CANCELLED, RT_OPDEL
transmitter_pid	int	PID of the communication process handling the transmission
msg_report	msg_rpt	Information about a message report object
text	flist_char	Contents of the report

Table 5-1 Message Object Fields (Continued)

Field Name	Type	Description
msgtype	wrapstring	Report type
channel	wrapstring	Relevant channel
contents	int	Bitmask field
datestr	wrapstring	Date of the report
alerts	flist_alerts	List of alerts generated
alert_class	wrapstring	Alert class
name	wrapstring	Alert name
msg	wrapstring	Message text
key	wrapstring	MSN or channel name
time	long	Time that the alert was generated
channel	wrapstring	Relevant channel
variables	flist_FILENAME_VAR	Information about the filename variables available for use
name	wrapstring	Name by which the variable is referenced
value	wrapstring	Value associated with the variable
identifier	int	Route identifier with which the variable is associated
target_chan	wrapstring	Used for routing/rerouting message objects

Table 5-2 Message Object err_type/error_expansion Values

Value of err_type CSC Mask	List of error_expansion Values
DEC_MASK	“General Partition Error” “Partition Error. Possible incorrect length of 1st seg.” “Segterm Partition Error” “Segment Incomplete Partition Error” “Start of Message Partition Error” “ISA segment out of order” “ISA segment parse error” “IEA segment out of order” “IEA segment parse error” “ISA/IEA numbers don’t match” “Incorrect GS count in IEA segment” “GS segment out of order” “GS segment parse error” “GE segment out of order” “GE segment parse error” “GS/GE group cutler numbers don’t match” “Incorrect ST count in GE segment” “ST segment out of order” “ST segment parse error” “SE segment out of order” “SE segment parse error” “ST/SE transaction numbers don’t match” “SE has incorrect segment count” “BQT segment out of order” “BQT segment parse error” “BQR segment out of order” “BQR segment parse error” “BEG segment out of order” “BEG segment parse error” “BCO segment out of order” “BCO segment parse error” “PO1 segment out of order” “BIG segment out of order” “BIG segment parse error” “Segment out of order” “Message Incomplete (ISA/GS/ST still active at end of message)” “Message ISA count = 0” “Invalid ST_Segment” “Invalid DUNS value” “Bad 838 version number”

Table 5-2 Message Object err_type/error_expansion Values (Continued)

Value of err_type CSC Mask	List of error_expansion Values
ROUTE_MASK	"No Primary Route Available." "No Reply Route Available." "Source Channel Does Not Exist." "Unknown Filename Variable"
XLATE_MASK	"Failed TPDB lookup" "Invalid UDF error" "Empty Message" "Source Channel Does Not Exist."
OUT_COMMS_MASK	"Failed SEGTERM conversion." "Failed on Addressee List or Content." "Cleo ASCII data rejection."
ACK_MASK	"997 Negative Acknowledgment" "997 Unable to Correlate"
OP_MASK	"Operator Delete from Channel Queue"

5.2 Communications

The Communications CSC is responsible for managing connections between ECPN and remote systems. The ECPN CSCI manages communications using the channel concept. Each remote site that ECPN connects to is designated as a separate channel. The channel contains information about connecting to a site, such as communications protocol, message type (X12 or UDF), connection frequency, and communications-specific information (e.g., email address for an email channel).

The Communications CSC consists of the following SCSCs:

- EditChannels
- Comms
- FTP Sessions
- File Transfer Protocol Daemon (ftpd)
- Electronic Send Electronic Mail (email_meta/email_send)
- Electronic Mail Daemon (emaild)
- Channel Status
- Incoming X12 Queue
- Outgoing Communication Queues
- Channel Database

5.2.1 EditChannels

The EditChannels application provides a GUI for adding, deleting, and modifying communication channels in the channel database and displaying the current channel state as set by the ECPN administrator. The channel database (described in Section 5.2.10) defines the operating characteristics of each communication interface. (EditChannels accesses the channel database via a socket interface to the RPCServer, as described in Section 5.1.1). The RPCServer actually performs modifications to the channel database as requested by EditChannels.

5.2.2 Comms

The comms process is started at system startup and is responsible for scheduling and executing communications sessions for FTP, ZMODEM, Kermit, and CLEO channels. It keeps an updated schedule of session times for all active channels and manages the child processes that handle individual sessions.

5.2.2.1 Scheduling Sessions

At startup, the comms process generates a “schedule” of sessions for all *active* channels (excluding email channels). Each schedule entry contains the channels database record number and a scheduled start time. The schedule is kept in ascending order, based on start time. Thus, the next communication session to start is always the first entry in the schedule. (Schedule entries are added when a channel is activated or a communication session starts. Schedule entries are deleted when a channel is deactivated or a communication session ends.) After each schedule modification, the timer indicating when the next session is to start is reset.

To view the schedule, send a SIGUSR2 to the parent comms process. The schedule will be printed in the comms debug log.

5.2.2.2 Communications Sessions

The comms process uses child processes to execute communication sessions. If there are no child processes currently running, or if all child processes are busy executing sessions, a new child process is spawned to execute the communication session. Otherwise, the first “waiting” child process is instructed to execute the communication session. An unlimited number of children may be spawned. However, if a child process remains inactive for a certain amount time (as defined by the registry entry “comms.ChildCleanupInterval”), it will be terminated with a SIGTERM signal by the parent comms process.

Upon completion of a communication session, the comms child process notifies the comms parent process via an event queue. The comms parent then schedules the next session (if the channel is still on) to start at the current time plus channel cycle time (in seconds). If the scheduled time is outside the communications window for the channel, the next session is scheduled to start at the beginning of the communications window for the following day.

If a communications session fails to connect, then the comms child will keep trying the connection until the number of retries set for the channel is exhausted. Between each retry attempt, the comms child will wait the specified retry_interval. If all retry attempts fail, then an alert is generated (either “FTP CONNECT” or “DIAL FAILED”). The comms parent is then notified that the session is complete, and the next session is scheduled.

5.2.2.3 Comms Children Database

Comms uses an RPC database, CommsChildren, to communicate between the parent and child processes. The CommsChildren database contains exactly one entry for each comms child process. Each entry contains the child's PID and the channel database record number for the communication session for which that child is currently executing. If the child is currently "waiting" (i.e., idle), the channel database record number will be '-1'. To view the entries in the CommsChildren database, run "CommsChildrenDB_text" from the command line.

When a child comms process is spawned, it reads the channel database record number from the CommsChildren database to determine the channel for which it is to execute a session. If the channel is a ZMODEM, Kermit, or CLEO channel, a serial session is run. If it is an FTP channel, an FTP session is run.

5.2.2.4 Serial Sessions

Before a serial session is started, the comms process selects an available device from those listed in the KermitDevices or CLEODevices files. If no device is available, the channel will wait until one becomes available. Availability of devices is controlled through the use of semaphores.

Once a device has been selected, ReadOutChnlQueue() is called iteratively to build a list of files to transmit. Each call to ReadOutChnlQueue() call returns a single file formatted for output. For multiple mode channels (e.g., Kermit, ZMODEM), this process consumes (at most) one channel queue record. For batch mode channels (e.g., CLEO), this process consumes all records on the queue that match the batch criteria (e.g., source channel cross-reference) or stops after the size of the file reaches the byte limit set in the channel's configuration. Each queue record is processed as follows:

- Using the route identifier from the channel queue record, the process gathers all routes that have a matching identifier and a status of "not complete".
- For an X12 transmit channel, the routes are used to identify the GSs and STs in the X12 message that are intended for transmit on the target channel. These GSs and STs are placed into ISA/IEA wrapped messages as follows:
    - The ISA/IEA wrappers from the original received X12 are used.
    - Field delimiters, separators, and/or sub-element separators are replaced as defined in the segment terminator replacement for the channel.
    - The ISA05/06 and ISA 07/08 fields are replaced with the values specified in the channel configuration record.
    - If specified in the channel configuration record, the file content is blocked into fixed length records (new line separated).

    - For CLEO channels set to transmit ASCII only, the message content is searched for characters that do not map from ASCII to EBCDIC (Extended Binary Coded Decimal Interchange Code) or that cause adverse behavior in transmit stream handling by the 3780 protocol. If these characters are found, the associated channel record is dequeued, and the message object is placed in the error queue.

- For a UDF channel, the file is returned as identified in the channel queue record. If the channel is a CLEO channel set to ASCII transfer mode, the file is first parsed for unprintable or non-whitespace characters. If these characters are found, the associated channel record is dequeued, and the message object is placed in the error queue.

Comms invokes either the kermit program (for Kermit and ZMODEM channels) or the 3780Plus program (for CLEO channels), using the script and initialization file for the channel. The steps performed by comms depend on each channel's modem script. For Kermit and ZMODEM channels, comms determines success or failure by processing data written to STDOUT by the kermit program while it processes the script. For the list of output strings processed from the Kermit and ZMODEM scripts, see Section 5.2.2.5. For CLEO channels, the exit status of the 3780Plus program determines the success or failure of the file transfer.

Once the communications session is complete, comms removes the successfully transmitted messages from the channel's outgoing queue and updates the status of the messages in the message log and channel log. If a message fails transmission, comms leaves the message in the outgoing channel queue for the next cycle and does not update the message log or channel log.

Comms then checks the incoming (InRaw) directory for messages received during the communications session. If any messages are found, they are logged in the ChannelLog and then placed in the incoming X12 or translation queue, depending on the message type supported by the channel.

Comms generates alerts for communication problems via the standard API, proc-alert().

5.2.2.5 Kermit

The kermit program is invoked by comms when a communications session for either a Kermit or ZMODEM channel is started. The functions that kermit performs depend on each channel's script, which should at a minimum, consist of these components: login, password, download files, upload files, and exit. File download success is determined by receiving the following string from STDOUT: "<filename> Successfully." File upload success is determined by signals inherent in the Kermit protocol.

Connection, authentication, file transmit, and file receive actions are performed by the kermit program, which is driven by a job script. The status of the connection and the status of each transmitted file is determined by parsing the output of the kermit program while executing the job script. Table 5-3 details the strings parsed and the actions taken during this parse operation. The existence of files in the local receive directory after a connection denotes a successful receive. The files are queued to either the incoming X12 queue or the translation queue, depending on the message type supported by the channel.

Table 5-3 Kermit Operation String/Action Relationship

String	Action
"Successfully"	The prior string is assumed to be a successfully sent file, and all messages contained within are marked as successfully transmitted and dequeued from the channel.
"send Unsuccessful"	The prior string is assumed to be an unsuccessfully sent file, and all messages contained within are left queued to the channel queue.
"NO CARRIER"	An alarm is set for 60 seconds. If after 60 seconds, the kermit process is still executing, the job is assumed to be hung, so it is terminated. All messages contained within the transmit files that have not been reported as successful are left queued to the channel.
"line for INPUT"	The process status is treated as a dial failure, and all messages contained within the transmit files are left queued to the channel.
"Cannot dial phone"	The process status is treated as a dial failure, and all messages contained within the transmit files are left queued to the channel.
"ABORT"	The process status is treated as a dial failure, and all messages contained within the transmit files are left queued to the channel.

Kermit channels support both batch and multiple file transfer. Filenames are constructed from variables and characters as follows:

Variables (within braces {}):

- `{jul} -` Julian Date (`001` to `366`)
- `{hr} -` Hour (`00` to `23`)
- `{min} -` Minute (`00` to `59`)
- `{sec} -` Second (`00` to `59`)
- `{time} -` hour and minute (`00` to `23`) (`00` to `59`)
- `{mon} -` Month (`01` to `12`)
- `{day} -` Day (`01` to `31`)

- `{year} -` 4-digit year (e.g., `1997`)
- `{yr} -` 2-digit year (e.g., `97`)
- `{sxrf} -` Channel reference for the message's source
- `{drxf} -` Channel reference for the message's destination
- `{c} -` Counter number (up to 8 digits)
- `{cent} -` 2-digit century (e.g., 21)

Note that additional variables may be valid for a specific message type. (For example, the {saacons-sid} variable is valid if SAACONS is the selected message type.) For a list of valid variables for a particular message type, see the DESCRIPTION box of the TRANSLATION tab of the edit channel window (described in the *Software User's Guide for Electronic Commerce Processing Node).*

Characters:

Other than variables and the braces that enclose those variables (`{}`), only alphanumeric characters and the following symbols may be entered in a file name. (Spaces are not allowed.)

- Hyphen (-)
- Period (.)
- Underscore (_)

The following fields can be used to configure a Kermit Channel:

Table 5-4 Kermit Channel Fields (AsyncStruct)

Field Name	Type	Description
device	char [DEVICE_LEN]	Modem device
recv_packet_length	int	Size of receive packets, in bytes
send_packet_length	int	Size of transmit packets, in bytes
baud_rate	int	Modem baud rate
char_size	int	7 or 8 bits
parity	enum ParityType	Possible Values: None, Even, Odd, Mark, Space
window_size	int	File transaction packet window size
dial_timeout	int	Dial timeout in seconds
block	int	Error checking level
byte-limit	int	Maximum size per batch file
receive_eop	int	Character to specify end-of-packet
escape_char	int	Character to specify escape character
phone	char [MAX_PHONE_LEN]	Phone number to dial

Table 5-4 Kermit Channel Fields (AsyncStruct) (Continued)

Field Name	Type	Description
outfile	char [MAX_OUTFILE_LEN]	Name of file to transmit
login	char [MAX_LOGIN_LEN]	Login ID to use
passwd	char [MAX_PASSWD_LEN]	Password to use
indir	char [MAX_INDIR_LEN]	Incoming directory
outdir	char [MAX_OUT_DIR_LEN]	Outgoing directory

5.2.2.6 ZMODEM

ZMODEM is invoked within a Kermit script by comms. Unlike Kermit, ZMODEM performs the upload and download of files using the rz and sz programs respectively. All other functions are the same as those for a Kermit channel (described in Section 5.2.2.5).

5.2.2.7 CLEO

CLEO (3780Plus) is invoked by comms when a communications session for a CLEO channel is started. The functions that CLEO performs depend on each channel's script, which should at a minimum, consist of these components: login, password, download files, upload files, and exit. The status of CLEO transmissions is based on the return codes CLEO sends to STDOUT. If no error code is returned and the session ends normally, transmitted messages may be assumed successful.

The status of CLEO file transmission is a function of the exit status of the 3780Plus execution. An exit status of zero denotes success, and all messages contained within the single transmit file are marked as successfully transmitted and are then dequeued from the channel queue. Any other exit status indicates failure, and all messages contained within the single transmit file remain queued to the channel. The existence of a file in the local receive directory after a connection denotes a successful receive. The received file is queued to either the incoming X12 queue or the translation queue, depending on the message type supported by the channel.

CLEO channels support only batch file transfer with a filename assigned in the edit channel window.

Table 5-5 CLEO Channel Fields (CleoStruct)

Field Name	Type	Description
device	char [DEVICE_ LEN]	Modem device for the channel
repeat_limit	int	See 3780Plus User's Guide, Chapter 5
rexmit_limit	int	See 3780Plus User's Guide, Chapter 5
wait_limit	int	See 3780Plus User's Guide, Chapter 5
delay_limit	int	See 3780Plus User's Guide, Chapter 5
terminal_type	enum TerminalType	See 3780Plus User's Guide, Chapter 5
compress_space	int	See 3780Plus User's Guide, Chapter 5
xmit_blocking_factor	int	See 3780Plus User's Guide, Chapter 5
modem_type	int	See 3780Plus User's Guide, Chapter 5
suppress_new_line	int	See 3780Plus User's Guide, Chapter 5
protocol	enum CleoProtocol	See 3780Plus User's Guide, Chapter 5
byte_limit	int	See 3780Plus User's Guide, Chapter 5
xmit_record_size	int	See 3780Plus User's Guide, Chapter 5
bid_limit	int	See 3780Plus User's Guide, Chapter 5
recv_limit	int	See 3780Plus User's Guide, Chapter 5

5.2.3 FTP Sessions

Scheduled FTP sessions exist for outgoing FTP communications only. (Note that the file transfer protocol daemon, ftpd, discussed in Section 5.2.4, manages the receipt of FTP files that are pushed to ECPN.) FTP sessions are executed by a comms child process each time a scheduled session is started for an FTP channel. FTP sessions call ReadOutChnlQueue() iteratively to build and transmit files.

Each call to ReadOutChnlQueue() returns a single file formatted for output. Multiple-mode file creation consumes (at most) one channel queue record. Batch mode file creation consumes all records on the queue that match the batch criteria (e.g., source channel cross-reference) or stops after the size of the created file reaches the byte limit for the channel. Each queue record is processed as follows:

- Using the route identifier from the channel queue record, the process gathers all routes that have a matching identifier and a status of “not complete”.
- For an X12 transmit channel, the routes are used to identify the GSs and STs in the X12 message that are intended for transmit on the target channel. These GSs and STs are placed into ISA/IEA wrapped messages as follows:
  - The ISA/IEA wrappers from the original received X12 are used.
  - Field delimiters, separators, and/or sub-element separators are replaced as defined in the segment terminator replacement for the channel.
  - The ISA05/06 and ISA07/08 fields are replaced with the values specified in the channel configuration record.
  - If specified in the channel configuration record, the file content is blocked into fixed length records (new line separated).
- For a UDF channel, the file is returned as identified in the channel queue record.

If there are no messages in the channel’s outgoing queue and the channel is set to push only, comms does not initiate a connection but will wait until the next session and check again. At that time, if there are messages queued for the channel, comms initiates an FTP connection using the site-specific information contained in the channel database (described in Section 5.2.10). If set to push/pull, comms will first upload (pull) remote files destined for ECPN. Upon successful retrieval and local queue storage of each file, comms will remove the file from the remote system. Comms will then download (push) any files destined for the remote system.

After each successful file send, comms deletes the successfully sent message(s) contained in the file from the channel’s outgoing queue and updates the status of the message(s) in the message log and channel log. If a message fails transmission, comms leaves the message in the outgoing channel queue for the next cycle and does not update the message log or channel log. Uploaded files are placed in either the incoming X12 queue or the incoming translation queue, depending on the channel’s message type.

Comms performs the following steps during each communications session:

1. Connects to the remote site’s IP address using the login provided (USER <username>).
2. Provides password and/or account, in any order, as prompted (PASS <password> and ACCT <account>). Note that comms will issue the password and account up to five times as prompted.
3. Sets the file transfer type as ASCII or binary.

4. Issues a change to the working directory, if necessary (CWD).

When pulling files on an FTP channel, comms performs the following functions:

5. Obtains a list of files in the pull directory, using full path (NLST) . If local globbing is set, this action returns *all* files. If remote globbing is set, only those files matching the specific remote mask are returned.

6. Processes the list, filtering out listing headers and local and parent directory tokens, and then builds the full path/name of each file to pull.

7. Receives the files from the remote system, using the full path (RETR).

8. After the file is received, confirmed, and locally stored and queued, deletes the file on the remote system (DELE). If "delete from pull dir" is configured, the pull directory/container is also removed from the remote system.

When pushing files on an FTP channel, comms performs the following functions:

9. Sends any site commands entered in the channel configuration window (described in Section 5.2.1).

10. If the trigger mode is TRG_LOCK, transfers the trigger lock file.

11. Transfers the data file as the generated filename using STOR or APPE, depending on the append or push setting. If the trigger mode is TRG_RENAME, transfers the data file as the generated trigger name and then renames the sent data file to the generated data file name.

12. If the trigger mode is TRG_CREATE, transfers the trigger file after the data file transfer is complete.

Once FTP file transfer is complete, comms:

13. Logs out of the remote system (QUIT).

FTP sessions support both multiple and batch file transfer. Filenames are constructed from variables and characters as follows:

Variables (within braces {}):

- `{jul} -` Julian Date (`001` to `366`)
- `{hr} -` Hour (`00` to `23`)
- `{min} -` Minute (`00` to `59`)
- `{sec} -` Second (`00` to `59`)
- `{time} -` hour and minute (`00` to `23`) (`00` to `59`)
- `{mon} -` Month (`01` to `12`)
- `{day} -` Day (`01` to `31`)
- `{year} -` 4-digit year (e.g., `1997`)

- `{yr} -` 2-digit year (e.g., `97`)
- `{sxrf} -` Channel reference for the message's source
- `{drxf} -` Channel reference for the message's destination
- `{c} -` Counter number (up to 8 digits)
- `{cent} -` 2-digit century (e.g., 21)

Note that additional variables may be valid for a specific message type. (For example, the {saacons-sid} variable is valid if SAACONS is the selected message type.) For a list of valid variables for a particular message type, see the DESCRIPTION box of the TRANSLATION tab of the edit channel window (described in the *Software User's Guide for Electronic Commerce Processing Node).*

Characters:

Other than variables and the braces that enclose those variables (`{}`), only alphanumeric characters and the following symbols may be entered in a file name. (Spaces are not allowed.)

- Hyphen (-)
- Period (.)
- Underscore (_)

If a message fails transmission, comms leaves the message in the outgoing channel queue for the next cycle. If a message is sent successfully, comms removes the message from the outgoing channel queue and updates both the message log and channel log to reflect successful transmission.

Table 5-6 FTP Channel Fields (FTPStruct)

Field Name	Type	Description
hostname	char [MAX_HOSTNAME_LEN]	Local host name for channel
login_id	char [MAX_LOGIN_LEN]	Remote login ID
password	char [MAX_PASSWD_LEN]	Remote password
workdir	char [MAX_DIR_LEN]	(Optional) Directory to cd to for pushing/pulling files
in_dir	char [MAX_DIR_LEN]	Remote download directory
out_dir	char [MAX_DIR_LEN]	Remote upload directory
filemask	char[MAX_OUTFILE_LEN]	String to be sent to remote system (if remote globbing set), or used locally (if local globbing set to filter selection of files to receive)
trigger_convention	enum TriggerStyle	None, Rename, Lock, or Create
trigdir	char [MAX_DIR_LEN]	Remote trigger directory/name

Table 5-6 FTP Channel Fields (FTPStruct) (Continued)

Field Name	Type	Description
site_cmd	char [MAX_SITE_CMD_LEN]	String to be sent as site command before each file transfer to set values such as record length
glob_option	int	Whether to do file name expansion, and if so, whether to do it locally or remotely
account	char [MAX_ACCOUNT_LEN]	Account information
appe_on_push	int	Use append command vice store command on transfer
del_pulldir	int	Whether to delete the pull directory/container after pull
byte_limit	int	Up to MaxInt. Max size of a batch file to be sent

To generate alerts for communication problems, comms connects to the alert daemon. (For a complete list of alerts, see Appendix A.)

5.2.4 File Transfer Protocol Daemon (ftpd)

The ftpd processes messages that are pushed to ECPN. No remote system is allowed to pull data from ECPN. The daemon is responsible for managing the connection request by the remote system and restricting the allowable functions to only those necessary to transfer files to ECPN. The ftpd establishes a “jail,” so that a remote user may not issue a change directory to any directory above the login directory. (The login directory is /h/data/global/EC/Messages/InRaw/ <Channel Name>.) Once the remote site has finished uploading the files, the ftpd passes the messages to the incoming X12 or translation queue, depending on the channel’s message type setting. To handle simultaneous FTP requests, the daemon can fork multiple processes.

5.2.5 Email Send Electronic Mail (email_meta/email_send)

The ability to transmit files via email is handled by two processes: email_meta and email_send. The Router places all outgoing message objects for all email channels on a single “meta” queue. The email_meta process consumes the single (meta) email queue and generates the email domain queues and the email stats database, based on the message destination domains. These email_send processes use the email_stats database and record locking to coordinate the consumption of the different domain queues.

The addressee list in an email channel configuration can be a list of several different email addresses, separated by commas. Each address has a username, and an @ symbol, followed by a domain name. The example addressee list—johndoe@acme.com, janedoe@widgets.net, ecpn@tools.acme.com, joeshmo@widgets.net—actually contains four addresses and three domains (acme.com, tools.acme.com, and widgets.net). With this hypothetical addressee list for channel EMAIL1, each message sent out channel EMAIL1 will actually be sent as three separate files, each to a different domain. (The file sent to widgets.net will have two addressees).

It is the responsibility of the email_meta process to take a message bound for the hypothetical EMAIL1 channel and vector it to the three different domain queues for processing by the email_send processes. The email_meta process also creates and manages the email_stats database. This database serves the dual purpose of feeding the outgoing email queue viewer application (described in Section 5.4.10), as well as the email_send processes. The email_send processes use record-level locking on the database entries to coordinate consumption among themselves. For a description of the email stats database, see Table 5-7.

Table 5-7 Email Stats DB Fields (EMAIL_STATS)

Field Name	Type	Description
domain	char [DB_MAXEMAILLEN]	Domain name, which is everything after the @ symbol in an email address
num_in_queue	int	Number of records in the corresponding domain queue
last_tot	unsigned long	Last successful transmit for the domain
last_attempt	unsigned long	Last transmit attempt for the domain
state	int	Busy, idle
on_off	int	Whether domain is on or off
cycle_thresh	int	Number of failures before alerting
thresh_enabled	int	Whether connection failure alert is on or off

The email_send processes are responsible for consuming the email domain queues by packaging and sending the queued messages in email files using the SMTP protocol. MIME is an optional field in the EDIT EMAIL window. If MIME is selected, each message is sent as a separate, base-64 MIME encoded attachment. Sites that wish to send and receive interchanges as MIME attachments must notify ECPN in advance. For MIME messages, the content-type will be application/edi-x12. With each attempt to process a queue entry, the email_send process also updates the following email stats database fields: Last TOT, Last Attempt, num_in_queue, and state.

The correlation between the email channels configuration records and email transmit processing, as well as the email queueing logic is described below:

- The email_meta process dequeues a record from the email meta queue, and constructs a list of domains to which the message must be sent using these steps:
    - Retrieves the route using the route id from the email queue record.

- - Retrieves the channel record associated with the route using the channel name stored in the route.
  - Constructs the domain list from the channel record's "send to" field.
  - Adds the route domain to the domain list if it is not already listed.
- The email_meta process enqueues the record to each of the targeted domain queues.
- The email_send process loops, searching the email stats database for a domain that is not locked and has not been attempted in the last "DOWN_HOST_RETRY" seconds.
- The email_send process calls ReadOutChnlQueue() to dequeue a record from the domain queue and build a file for transmit. (Note that email channels send in multiple mode only, so batch mode is not permitted.) ReadOutChnlQueue() then performs the following processing:
  - For an X12 channel, the routes are used to identify the GSs and STs in the X12 message that are intended for transmit on the target domain. These GSs and STs are placed into ISA/IEA wrapped messages as follows:
    - The ISA/IEA wrappers from the original received X12 are used.
    - Field delimiters, separators, and/or sub-element separators are replaced as defined in the segment terminator replacement for the channel.
    - The ISA05/06 and ISA07/08 fields are replaced with the values specified in the channel configuration record.
    - If specified in the channel configuration record, the file content is blocked into fixed length records (new line separated).
  - For a UDF channel, the file is returned as identified in the channel queue record.
- The route is passed back from ReadOutChnlQueue(). The email_send process uses the channel name in the route to access the channel configuration and determine whether to MIME encode the message.
- The email_send process(es) query the host denoted by the domain name to get a list of SMTP servers for the domain.
- The email_send process makes an SMTP connection with an SMTP server from the list, negotiates the from/to addresses, and then sends all of the data in the domain queue.

- The email_send process calls UpdateOutChnlQueue() to update the route's status within the sent message object and to delete the associated record from the domain queue. Note that for data send errors, the records are left on the queue for subsequent attempts. For addressee or SMTP server negotiation errors, the message object is marked with an error, sent to the error queue, and dequeued from the domain queue.

Table 5-8 Email Channel Fields (EmailStruct)

Field Name	Type	Description
mime_capable	int	Flag that denotes whether a channel sends messages as MIME attachments
send_to_addr	char [MAX_ADDR_LEN]	List of addresses to which messages routed to the channel should be sent. Note that this could be a list that contains multiple domains.
recv_from_domain	char [MAX_ADDR_LEN]	List of addresses or partial addresses used by the emaild process (described in Section 5.2.6) to attribute received email to a specific channel

5.2.6 Electronic Mail Daemon (emaild)

The emaild SCSC is responsible for processing messages received via SMTP email. The emaild process checks for incoming messages in the ecedi mailbox. Received messages are recorded in the channel logs, stripped of addressing header information, and passed to either the incoming X12 queue or incoming translation queue for processing. If emaild receives a message with a MIME attachment, it decodes each attachment and treats it as a separate message. Emaild can decode MIME attachments that use one of the following content transfer encodings:

- 7 bit
- 8 bit
- binary
- quoted-printable
- base-64

The emaild process searches the "From" address lists in each of the channel configuration records (described in Table 5-8) to attribute a received email message to an established channel. When a message is received and no active channel is found with a partial or full matching from address, the message is placed into the RejectedEmail box. It can be viewed and reinjected using the RejectedEmail application (described in Section 5.4.12).

5.2.7 Channel Status

The Channel Status application (ChanStats) provides a static view of the communication channels configured in the channel database (described in Section 5.2.10). The last time of receipt, last time of transmit, current outgoing message backlog, and current channel status are provided for each channel in sortable columns.

Table 5-9 Channel Status Database (CHAN_STAT_REC)

Field Name	Type	Description
status	enum IfaceStatType	Status of the channel (e.g., DOWN, IDLE, BUSY)
tor	long	Last time of receipt
tot	long	Last time of transmit
backlog	int	Number of messages in an outgoing channel's queue
db_rec	int	Database record number. This applies to both the channels database and the chanstat database, since these two databases must always match.

5.2.8 Incoming X12 Queue

The incoming X12 queue contains a record for each file to be processed by the Router (described in Section 5.3). Once the incoming file is processed by the Router, the record is removed from the queue. This queue is populated by the incoming communication channels that are designated as X12 channels. In addition, this queue is populated by the incoming UDF to X12 Translator and the error queue, message log, and channel log applications. The queue has no fixed capacity and is limited only by available disk space.

The Incoming X12 Queue Viewer (described in Section 5.4.4) provides a text-based interface for displaying the current content of the queue. The content of the incoming X12 queue is described in Table 5-10.

Table 5-10 Incoming X12 Queue Fields (IN_X12_FILE_REC)

Field Name	Type	Description
IN_COMMS_REC	Variant of the union	Normal X12 received file processing record. Contains the following pieces (indicated by indention):
x12_filename	SHORT_FILE_NAME	Local file containing the received X12
pull_filename	SHORT_FILE_NAME	Name of the file on the remote system when it was pulled
in_chan_name	CHAN_NAME	Name of the incoming channel
TOR	u_long	Time the message was received
IN_UDF2X12_REC	Variant of the union	Files received on a UDF channel. Contains the following pieces (indicated by indention):
genx12_and_824	UDF2X12_REC	Structure for the UDF to X12 Translator. (See Table 5-11.)
pull_filename	SHORT_FILE_NAME	Name of the file on the remote system when it was pulled
in_chan_name	CHAN_NAME	Name of the incoming channel
TOR	u_long	Time of receipt
UDF2X12_REXLATE_REC	Variant of the union	Files that need to go back through the Translator. Placed in the IN_X12_FILE_Q by the RPCServer in response to a REXLATE action taken by the user in the error log or message log. Contains the following pieces (indicated by indention):
msn	MSN_NAME	MSN being reprocessed
genx12_and_824	UDF2X12_REC	Structure from the UDF to X12 Translator
X122UDF_REC	Variant of the union	Placed in the IN_X12_FILE_Q by the X12 to UDF Translator. It represents the generated data for a message being routed out a UDF channel. Contains the following pieces (indicated by indention):
msn	MSN_NAME	MSN being translated
identifier	int	Identifier for the routes in parent_msn

Table 5-10 Incoming X12 Queue Fields (IN_X12_FILE_REC) (Continued)

Field Name	Type	Description
ack_filename	SHORT_FILE_ NAME	File generated by the X12 to UDF Translator
xltr_error_type	short	Error code from outgoing translation. TPDB_ERR - Problem finding the TPDB information X12_ERR - Invalid or corrupt X12
out_chan_name	CHAN_NAME	Outgoing channel name passed in by the Translator
REROUTE_REC	Variant of the union	Placed in the IN_X12_FILE_Q by the RPCServer in response to a REROUTE action taken by the user in the error log or message log. Contains the following pieces (indicated by indention):
msn	MSN_NAME	MSN being reprocessed
TRANSMIT_REC	Variant of the union	Not currently implemented
RETRANSMIT_REC	Variant of the union	Placed in the IN_X12_FILE_Q by the RPCServer in response to a RETRANSMIT action taken by the user in the outgoing channel log application. Contains the following pieces (indicated by indention):
out_chan_name	CHAN_NAME	Name of the channel on which to retransmit
mod_name	MOD_NAME	Name of the ECPN administrator selecting RETRANSMIT from the channel log application. This is used to add an audit event to the message object.
mod_host	MOD_HOST	Host from which RETRANSMIT was selected on the channel log application. This is used to add an audit event to the message object.
index	int	Index to the channel log
date_str	DATE_STR	Date of the channel log
MSG_REPORT_REC	Variant of the union	Placed on the IN_X12_FILE_Q by the Message Reporter in response to a crontab invocation to generate message reports
contents	u_int	Contents indicator
chan_name	CHAN_NAME	Channel for the report

Table 5-10 Incoming X12 Queue Fields (IN_X12_FILE_REC) (Continued)

Field Name	Type	Description
log_date	DATE_STR	Log date for the report
IN_SPS_EDA_REC	Variant of the union	Placed in the IN_X12_FILE_Q by comms channels of message type SPS-EDA
ps	SHORT_FILE_ NAME	Name of the incoming postscript file
ps_incoming_name	SHORT_FILE_ NAME	Name of the postscript file as sent by the remote system
ps_TOR	u_long	Time of receipt for the postscript file
idx	SHORT_FILE_ NAME	Name of the incoming index file
idx-incoming_name	SHORT_FILE_ NAME	Name of the index file as sent by the remote system
idx_TOR	u_long	Time or receipt for the index file
chan_name	CHAN_NAME	Name of the channel by which the file was received

Table 5-11 UDF to X12 Translator Structure (UDF2X12_REC)

Field Name	Type	Description
sectinfo_filename	SHORT_FILE_ NAME	File containing section info
udf_filename	SHORT_FILE_ NAME	File containing the UDF
x12_filename	SHORT_FILE_ NAME	Local file containing an X12 generated by the UDF to X12 Translator
gen824_filename	SHORT_FILE_ NAME	File generated by the Translator containing an X12 824 to be sent back to the sender
xltr_error_type	short	Status of the translation: TPDB_ERR - Problem finding the TPDB information UDF_ERR - Invalid or corrupt UDF
errfile	SHORT_FILE_ NAME	File generated by the incoming Translator which gives debug information concerning the translation
TOX	u_long	Time that the UDF to X12 Translator translated the file

5.2.9 Outgoing Communication Queues

An outgoing communication queue exists for each channel configured in the channel database (described in Section 5.2.10), except for email channels. For email channels, there is one large meta queue and various domain queues (as described in Section 5.2.5). The contents of an outgoing communication queue is described in Table 5-12. Currently, only a single precedence is used for all message traffic.

Entries to the outgoing communication queues are appended by the following means:

- The Router (described in Section 5.3.1) appends a message to the outgoing communication queues after it has parsed and decoded the message and determined which (if any) channels to route the message to. Because all message appended by the Router are in X12 format, the UDF Filename field is left empty.

- The Translator (described in Section 5.5.4) appends messages to the outgoing communication queues after it has successfully converted a message from an X12 to a UDF. The UDF Filename field is populated with the newly created UDF file.

Entries are removed from the outgoing communications queue only after they have been successfully transmitted. Messages that fail transmission will be left in the queue for subsequent channel cycles. Note that for single ST-based channels (1 ISA/GS/ST), an entry will remain in the outgoing communications queue until all of the STs created by the outgoing comms channel have been transmitted. The EC message object will contain a record of untransmitted STs to ensure that duplicate STs are not transmitted out a channel.

These queues have no fixed capacity and are limited only by available disk space.

Table 5-12 Outgoing Communication Queue Fields (OUT_CHAN_REC)

Field Name	Type	Description
msn_name	MSN_NAME	(format: NNNNNNNN/YYYYMDD)
out_udf_filename	SHORT_FILE_NAME	(For UDF channels only) UDF file to transmit
identifier	int	Route identifier designated for the destination channel

5.2.10 Channel Database

The channel database stores information specific to each communication channel entry. It is populated by user entry through the EditChannels application (described in Section 5.2.1) and is used to control core processing such as routing, translation, incoming communications, and outgoing communications. For a complete description of the channel database record, see Table 5-13. Each database record also contains interface-specific information. For a listing of these protocol-specific fields, see Table 5-4 (Kermit), Table 5-5 (CLEO), Table 5-6 (FTP), and Table 5-8 (email).

Table 5-13 Channel Database Fields

Field Name	Type	Description
chname	char [CHNAME_LEN]	Channel name
machine	char [MAXHOSTNAMELEN]	Local host for channel
intrfc	char [INTERFACE_LEN]	Possible values: FTP, Email, Kermit, ZMODEM, CLEO
message_type	char [MESSAGE_TYPE_LEN]	Possible values: X12ISA, SAACONS UDF, SPS UDF
node_type	enum NodeType	Possible values: GATEWAY, VAN, NEP, AIS
data_type	enum Data Type	Possible values: Binary, ASCII
state	enum ChannelState	Set to ON or OFF by user action through the EditChannels application
chkGS02.enabled	int	Indicator for whether to validate the GS02 field route
editor	char [EDITOR_LEN]	Name of executable invoked to edit the channel. Possible values: CleoConfig, FileConfig, EditKermit, EmailEdit.
recv	int	Channel capable of receiving
xmit	int	Channel capable of transmitting
transfer_style	enum XferStyle	Possible values: Batch, Multiple
hdr_trailor	int	Indicates whether to add headers and trailers
singleST	int	On or off for single ST transfer
isa_sender_id_qualifier	char [ID_QUALIFIER_LEN]	ISA05 replacement value
isa_sender_id	char [ISA_ID_LEN]	ISA06 replacement value
isa_recv_id_qualifier	char [ID_QUALIFIER_LEN]	ISA07 replacement value

Table 5-13 Channel Database Fields (Continued)

Field Name	Type	Description
isa_recv_id	char [ISA_ID_LEN]	ISA08 replacement value
conv	struct SegTermConv	
do_conv	int	Whether outgoing conversion is enabled
segterm	unsigned int [2]	Two-byte segment terminator
elemsep	unsigned int	Element separator
subelemsep	unsigned int	Subelement separator
collision detection	unsigned int	Whether to put a message in the error queue if a collision is detected
admin	struct AdminInfo	
adminpath	char [ADMIN_MAX_PATH_ LEN]	Email address or path to push admin files to
adminfname	char [ADMIN_MAX_PATH_ LEN]	File name to transmit as
record_len	int	Maximum length of a file sent to the remote system
xref	char [XREF_LEN]	Channel cross-reference string
remote_os	int	Operating system (OS) that remote system is running
remote_os_str	char [REMOS_STR_MAX]	OS as entered by the ECPN administrator
Interface specific information	union	For a listing of these protocol-specific fields, see Table 5-4 (Kermit), Table 5-5 (CLEO), Table 5-6 (FTP), and Table 5-8 (email).
begin_win	int	Beginning of connection window (in seconds past 00:00)
end_win	int	End of connection window (in seconds past 00:00)
days	int [7]	Array of days on which to connect (0=SUN, 1=MON, . . . 6=SAT)

Table 5-13 Channel Database Fields (Continued)

Field Name	Type	Description
cycle	int	Cycle length (in seconds)
retry_interval	int	Indicates how often to retry a failed connection
num_retries	int	Indicates how many times to retry a failed connection before quitting

5.3 X12 Message Processing

The X12 Message Processing CSC applies to those SCSCs responsible for parsing and routing X12 messages. This CSC consists of the following SCSCs:

- Regular Received X12 Handling
- Translated UDF to X12 Handling
- UDF to X12 Retranslate
- System Generated File Handling
- Reroute Handling
- Retransmit Handling
- Message Report Handling
- SPS-EDA Handling
- Parsing
- Route Lookup
- Queueing

The router process searches the incoming X12 queue for the next available IN_X12_FILE_REC record. This record can represent one of several functions, as defined by the variant record shown in Table 5-10. The format of the IN_X12_FILE_REC is also described in Table 5-10.

The following sections detail the handling of each variant of the IN_X12_FILE_REC.

5.3.1 Regular Received X12 Handling

This SCSC handles X12 messages received on an X12 channel. The file is partitioned into messages, and each message is parsed, routed, and queued in accordance with the methods described in Section 5.3.9, 5.3.10, and 5.3.11.

5.3.2 Translated UDF to X12 Handling

This SCSC handles records generated from the UDF to X12 Translator (described in Section 5.5.1). The message is parsed, routed, and queued in accordance with the methods described in Section 5.3.9, 5.3.10, and 5.3.11.

5.3.3 UDF to X12 Retranslate

This SCSC handles records placed in the IN_X12_FILE_Q by the RPCServer in response to a RETRANSLATE action taken by the user in the error log or message log. The original UDF content is extracted from the message object, and a record is queued to the UDF to X12 Translator (described in Section 5.5.1). This action results in the creation of a new message object for the retranslated message.

5.3.4 System Generated File Handling

This SCSC handles the system-generated X12 messages that result from translation (described in Section 5.5.1 and 5.5.4). Incoming translation may generate an 824 acknowledgment for translated messages. If so, the 824_997_filename field of the current IN_X12_FILE_REC record is populated with the filename containing the 824.

Using the IN_X12_FILE_REC's incoming channel field, the router looks up the channel configuration to check the acknowledgment configuration. If the channel is configured to send *all* acknowledgments, the 824 text is parsed through the Parse SCSC (described in Section 5.3.9). If the channel is configured to send acknowledgments only on failure *and* the IN_X12_FILE_REC's xltr_error_type field is set for error, the router will pass the 824 text to the Parse SCSC. Outgoing translation generates a 997 X12 message. For a description of 997 routing, see the Routing SCSC (described in Section 5.3.10).

5.3.5 Reroute Handling

This SCSC handles records placed in the IN_X12_FILE_Q by the RPCServer in response to a REROUTE action taken by the user in the error log or message log application (described in Section 5.4.5 and 5.4.6). Routes are not redundantly assigned. As a result, a reroute action only assigns and queues routes that are new since the last route attempt. The reroute action uses the MSN to access the message object to determine the routes that are and are not satisfied in accordance with the route database.

5.3.6 Retransmit Handling

This SCSC handles records placed in the IN_X12_FILE_Q by the RPCServer in response to a RETRANSMIT action taken by the user in the outgoing channel log application (described in Section 5.4.3). RETRANSMIT of a file from the channel log application results in the same content (MSN/GS/ST message portions) of the original file being flagged for routing out the same channel. The normal communication logic for queue/route management handles these added routes, so changes in channel configuration affect the repackaging of the messages into files.

5.3.7 Message Report Handling

This SCSC handles records (MsgReports) placed in the IN_X12_FILE_Q by the MsgReporter. The MsgReporter, run as a regularly scheduled cron job, is responsible for generating message traffic reports for individual communication channels. Message reports are logged like any other message, and routing is determined based on the communication channel's admin tab configuration. The message route database is bypassed completely.

5.3.8 SPS-EDA Handling

This SCSC handles records placed in the IN_X12_FILE_Q by the incoming communication channels of message type SPS-EDA. Files are expected to come in pairs, including an index file and a postscript file. The two files are treated as a sectioned message, with the postscript file being set as a section keeper. Only the postscript file is routed according to Section 5.3.10. Upon transmission, the other section—the index file—is gathered and sent with the postscript file.

5.3.9 Parsing

A file is parsed into one or more message objects by the parsing SCSC. Parsing identifies and validates relevant information in an X12 message and stores this information in message objects on a per ISA basis.

Precedence is assigned based on the message type and stored with the message object. 824s and 997s receive ACK precedence, while everything else gets ROUTINE precedence.

5.3.9.1 Element Validation

Some X12 segments are validated by checking to insure the elements in the segment comply with the X12 3040 standard. The Router validates elements by performing the following checks on each element:

- Length (minimum and maximum)
- Embedded blanks allowed
- Mandatory or optional
- Field type - valid types in accordance with X12 are:
    - ID
    - String
    - Date
    - Time
    - Numeric
    - Decimal Number

5.3.9.2 Element Storage

The following elements are parsed and stored in a message object on a per ISA basis.

- BEG
    - 01 Purpose
    - 02 Type Code
    - 03 Purchase Order Number

- BIG
  - 01 Invoice Date
  - 02 Invoice Number
  - 04 Purchase Order Number

- BQT
  - 01 Purpose
  - 02 Ref Num
  - 03 RFQ Date
  - 04 Date/Time Qual
  - 05 Date

- BQR
  - 01 Purpose
  - 02 Ref Num
  - 03 RFQ Date
  - 04 Date/Time Qual
  - 05 Date

- BCO
  - 01 Purpose
  - 02 Ref Num
  - 03 RFQ Date

- GS - All Fields (GS01 - GS09)
- GE - All Fields (GS01 - GS08)
- IEA - All Fields (IEA01 -IEA02)
- ISA - All Fields (ISA01 - ISA18)
- SE - All Fields (SE01 - SE02)
- ST - All Fields (ST01 - ST02)

5.3.9.3 Logging and Message Object Storage

Parsing errors for each message object are identified and stored in the message object. Each of the messages is also logged in the message log. If parsing errors exist, the message is also appended to the error queue; otherwise, the message has passed parsing.

If the parent_msn field of the IN_X12_FILE_REC is populated, then the parent_msn is placed in the newly created message object(s) as a linkage. The MSN(s) for the newly created message object(s) is also stored in the parent message object.

5.3.10 Route Lookup

Routes for all messages except 824s, 997s, and retransmitted records are determined based on the message route database. This database establishes a link between two circuits, routing messages from one to the other based on the following criteria:

- ISA/GS To - To specify that the system route only messages *addressed to* a certain site (using the value that appears in the ISA08 or GS03 field of the message). Note that this function can result in portions (GSs) of a message assigned to a route without the whole message being assigned to a route.

- VAN/Filename Pattern - To specify that the system route only messages whose original file name matches the specified file pattern. A file pattern may contain wild cards, such as "*", for matching purposes.

- GS01 - To specify that the system route to a channel only those incoming messages of a certain X12 transaction type.

- All-Channel - To specify that the system route all of the messages from the source channel to the specified destination channel.

A list of channels is provided for source and destination channel selection. Duplicate route entries, as well as individual entries where the source and destination are the same, are not allowed.

5.3.10.1 824 Routing

A system-generated 824 is a by-product of translating a UDF to an X12. The original source channel of the UDF contains the channel or email addresses to which the 824 should be routed. This route is assigned explicitly, bypassing the message route database.

5.3.10.2 997 Routing

997s are received as independent IN_X12_FILE_REC records. The in_chan_name field specifies where the 997 is to be sent. This route is assigned explicitly, bypassing the message route database.

5.3.10.3 Message Route Database

An entry in the message route database is defined by the source channel of the incoming message, a routing field, and the destination channel of the outgoing message. The routing can include criteria such as the receiver (TO) or the filename. The wildcard entry ALL also exists for the source channel and routing fields. If ALL is selected for these fields, all messages pass the message routing criteria for that entry.

Individual entries in the message route database can be activated or deactivated to control their current status during runtime checking.

Table 5-14 Message Route Database Fields (ROUTE_REC)

Field Name	Type	Description
active	char	Indicates if the entry is active or not (1=ON; 0=OFF)
type	char	Indicates the type of routing being used (e.g., file name, ISA/GS TO)
cc	char	Indicates whether the route is a primary route or a cc route (1=cc route; 0=primary route)
to	char [DB_ISAGS_TO_LEN]	ISA or GS receiver (to) ID
src_channel	char [DB_CHNAME_LEN]	Source channel, FROM field, TO field, or filename prefix by which to route. Also contains entry telling by which criterion to route.
id	char [DB_FILE_PATTERN_LEN]	Up to three-letter prefix of the received file
dest_channel	char [DB_CHNAME_LEN]	Contains the destination channel for the route. Also contains a destination email address when the outgoing channel is an email interface.
gs01	GS01_ENTRY [MAX_GS01]	GS01 value on which to route

5.3.11 Queueing

If the message is being routed to a UDF channel, an OUT_X12_FILE_REC is placed in the outgoing translation queue. If a message is being routed to any email channel, an OUT_X12_FILE_REC is placed in the single email meta queue. For all other X12 channel destinations, an OUT_X12_FILE_REC is placed in the queue for that channel.

5.4 Audit

The Audit CSC is responsible for creating and managing an audit trail for all messages processed by ECPN. The SCSCs of the Audit CSC are:

- Message Log Database
- Error Queue
- Channel Log
- Incoming X12 Queue Viewer
- Message Log Viewer
- Error Queue Viewer
- Journal Data Summary (JDS) Viewer
- Raw Message Viewer
- Channel Log Viewer
- Email Domain Queue Viewer
- Channel Queue Viewer
- Rejected Email Box Viewer
- RDBMS Injector
- RDBMS Retrieval
- RDBMS Message Database
- RDBMS Table Database

5.4.1 Message Log Database

The message log database is a collection of RPC-based daily message logs. Each log maintains a summary record for each message object contained within it. Each day, a new message log is created. This log has no fixed capacity and is limited only by available disk space and unique message sequence numbers (range 1 - 99,999,999). Each message object is maintained in a compressed data file stored in Daily/<yyyymmdd>/Archives/msg_objs.

The Message Log Viewer provides GUI interface access to the set of message logs and the content of each log. For a complete description of this GUI, see Section 5.4.5.

Table 5-15 Daily Message Log Database Fields (MSG_LOG_REC)

Field Name	Type	Description
msn_name	MSN_NAME	Message Sequence Number (format: SNNNNNNNN/YYYYMMDD). The numeric portion starts at 00000001 for the first incoming message of a day and is incremented sequentially, so that it represents the true order of decoded incoming messages to the system.
in_chan_name	CHAN_NAME	Name of the originating communications channel
in_filename	MLR_FILE_NAME	Incoming file name (UDF or X12 file)

Table 5-15 Daily Message Log Database Fields (MSG_LOG_REC) (Continued)

Field Name	Type	Description
TOR	long	Time that the incoming file was received on the communications channel
TOP	long	Time that the Router process completed processing the incoming file and created the message object
recv_id	ID_NAME	Receiver Interchange Name (X12 ISA08 field)
send_id	ID_NAME	Sender Interchange Name (X12 ISA06 field)
ic_number	int	ISA Interchange Control Number (X12 ISA13 field)
msg_size	int	Number of bytes in the message
error_mask	int	Identifies the type of error for messages in the error queue

5.4.2 Error Queue

The error queue contains summary information for each message that fails translation, decoding, routing, or conversion. It is a single queue that spans all operational days. The error queue is populated by the Router (as described in Section 5.3.1), the UDF to X12 Translator (as described in Section 5.5.1), the X12 to UDF Translator (as described in Section 5.5.4), and by outgoing communications processes (as described in Section 5.2) as follows:

- Incoming UDF messages that fail UDF to X12 translation because of semantic or syntax errors are flagged as an error by the UDF to X12 Translator and placed in the error queue by the Router process.
- Incoming UDF messages that fail UDF to X12 translation because of internal database lookup errors are flagged as an error by the UDF to X12 Translator and placed in the error queue by the Router process.
- Incoming X12 or post UDF to X12 translation messages that fail X12 decode because of semantic or syntax errors are placed in the error queue by the Router process.
- Incoming X12 or post UDF to X12 translation messages that do not have an outgoing route available in the route database (for any piece of the message) are placed in the error queue by the Router process.
- Outgoing UDF messages that fail X12 to UDF translation because of semantic or syntax errors are placed in the error queue by the X12 to UDF Translator process.
- Outgoing UDF messages that fail X12 to UDF translation because of internal database lookup errors are placed in the error queue by the X12 to UDF Translator process.

- Outgoing X12 messages that fail segment terminator conversion because of conflicts between the channel configuration record and the message content (target delimiter pre-exists in the outgoing message) are placed in the error queue by the outgoing communications process.

This error queue has no fixed capacity and is limited only by available disk space. The Error Queue Viewer (described in Section 5.4.6) provides a GUI for viewing and processing the queue entries.

Table 5-16 Error Queue Fields (MSG_LOG_REC)

Field Name	Type	Description
msn_name	MSN_NAME	Message Sequence Number (format: SNNNNNNNN/YYYYMMDD)
in_chan_name	CHAN_NAME	Name of the originating communications channel
in_filename	MLR_FILE_NAME	Incoming filename (UDF or X12 file)
TOR	long	Time that the incoming file was received on the communications channel
TOP	long	Time that the Router process completed processing the incoming file, and created the message object
recv_id	ID_NAME	Receiver Interchange Name (X12 ISA08 field)
send_id	ID_NAME	Sender Interchange Name (X12 ISA06 field)
ic_number	int	ISA Interchange Control Number (X12 ISA13 field)
msg_size	int	Number of bytes in the message
error_mask	int	Identifies the type of error for messages in the error queue

5.4.3 Channel Log

A channel log contains a record for each receipt or transmission on a channel. (Separate incoming and outgoing channel logs exist for each channel.) The actual text received or transmitted is maintained in a compressed text data file that is stored in Daily/<yyyymmdd>/ChannelLogs/<channel>/indata and Daily/<yyyymmdd>/ChannelLogs/<channel>/outdata.

A channel log has no fixed capacity and is limited only by available disk space. Channels logs are rolled over on a daily basis, so that a given log contains only the current day’s incoming or outgoing messages. For details about the Channel Log Viewer, see Section 5.4.9.

5.4.4 Incoming X12 Queue Viewer

The Incoming X12 Queue Viewer displays the records contained in the incoming X12 queue (described in Section 5.2.8). For the location of the data stored, see Section 5.4.3.

5.4.5 Message Log Viewer

The Message Log Viewer is implemented through the MsgLog application and its interface with the RPC message log databases. This GUI application provides the ECPN administrator with a tabular form of summary information for each message that is processed by the Router module (described in Section 5.3.1).

The Message Log Viewer interfaces with the RPCServer (described in Section 5.1.1) to obtain all message log information. The Message Log Viewer is able to connect to any reachable ECPN RPCServer across a TCP/IP network and obtain message log information stored on that host. For the location of the data stored, see Section 5.4.1.

The ECPN administrator can reroute or retranslate messages through the MsgLog application in accordance with the rerouting and retranslating functionality provided by the Error Queue Viewer application (described in Section 5.4.6). Rerouting or retranslating that message from the Message Log Viewer removes the message record from the error queue.

The tabular information provided by this application is a formatted version of the contents of the RPC-based message log. For a listing of these fields, see Table 5-15. The ECPN administrator can sort the records on any column in the table of records. The JDS Viewer (see Section 5.4.7) and the Raw Message Viewer (see Section 5.4.8) can be invoked for viewing individual message log entries. Message Log entries may also be annotated from the Message Log Viewer.

5.4.6 Error Queue Viewer

The Error Queue Viewer is implemented through the MsgLog application and its interface with the RPC-based error queue. This application interfaces with the RPCServer (described in Section 5.1.1) to obtain all error queue information. The Error Queue Viewer is able to connect to any reachable ECPN RPCServer across a TCP/IP network and obtain error queue information stored on that host.

The ECPN administrator has the ability to clear records from the error queue by deleting them. Note that deleting a record from the error queue does not remove the message from the system. The message is still in the message log database (described in Section 5.4.1) and can be viewed via the Message Log Viewer.

Because an ECPN administrator cannot modify messages, the following types of failed messages cannot be resolved by a reroute or retranslation. The only way to remove these messages from the error queue is to delete them.

- Incoming UDF messages that failed UDF to X12 translation because of semantic or syntax errors can be sent back to the UDF to X12 Translator with a retranslate operation. The message will fail again and be placed back in the error queue by the Router process.
- Incoming X12 or post UDF to X12 translation messages that failed X12 decode because of semantic or syntax errors are sent back to the Router process with a reroute operation. The message will fail again and be placed back in the error queue by the Router process.
- Outgoing UDF messages that failed X12 to UDF translation because of semantic or syntax errors are sent back to the Router process with a reroute operation. The message will fail again and be placed back in the error queue by the X12 to UDF translation process.

NOTE: Reroute and retranslate actions actually remove the message from the queue; but, because the message fails again, it is immediately placed back in the queue.

The ECPN administrator can clear the following message types from the error queue by rerouting or retranslating them:

- Incoming UDF messages that failed UDF to X12 translation because of internal database lookup errors are sent back to the UDF to X12 Translator with a retranslate operation. If the ECPN administrator has updated the database to resolve the initial lookup failure, the message will continue through processing. Otherwise, the message will fail again and be placed back in the error queue by the Router process.
- Incoming X12 or post UDF to X12 translation messages that did not have an outgoing route available in the route database (for any piece of the message) are sent back to the Router process with a reroute operation. If the ECPN administrator has updated the database with a route for the message, the message will continue through processing. Otherwise, the message will fail again and be placed back in the error queue by the Router process.
- Outgoing UDF messages that failed X12 to UDF translation because of internal database lookup errors are sent back to the router and then to the X12 to UDF Translator process with a reroute operation. If the ECPN administrator has updated the database to resolve the initial lookup failure, the message will continue through processing. Otherwise, the message will fail again and be placed back in the error queue by the X12 to UDF Translator process.

- Outgoing X12 messages that failed segment terminator conversion, due to terminators and separators in the message body, are sent back to the Router process with a reroute operation. If the ECPN administrator has updated the channel record to resolve the initial conflict, the message will continue through processing. Otherwise, the message will fail again and be placed back in the error queue by the outgoing communications process.

The tabular information provided by the Error Queue Viewer is a formatted version of the content of the RPC-based error queue. For a listing of these fields, see Table 5-16. The ECPN administrator has the ability to sort the records on any column in the table of records. The JDS Viewer (see Section 5.4.7) and the Raw Message Viewer (see Section 5.4.8) can be invoked for individual error queue entries. Error Queue entries may also be annotated from the Error Queue Viewer.

5.4.7 Journal Data Summary (JDS) Viewer

The JDS Viewer provides a common GUI for displaying the message content, identified message errors, and all associated data and actions for the message object. It is invoked from the Message Log Viewer (described in Section 5.4.5), the Error Queue Viewer (described in Section 5.4.6), the ObjectMMI (described in Section 5.4.14), the Email Domain Viewer (described in Section 5.4.10), and the outgoing channel logs. For the location of the data stored, see Section 5.4.1.

The message content is formatted in a pane with standardized graphic symbols in place of delimiters and with placeholder text for any binary data fields (X12 BIN02 fields). Identified errors are described in a separate error text pane. Previous and Next buttons are available for navigating through the highlighted errors in the message body. Associated data and actions are represented in yet another pane. The content of this pane is described in Table 5-17.

Table 5-17 JDS Viewer Message Journal Pane

Field Name	Description
General Info	Contains content descriptions that are applicable to the whole message, including: ISA Sender, ISA Receiver, IC Number, number of message segments, message size in bytes, received delimiters
Errors	Contains a list of message processing errors along with the message segment and offset values in which the errors were identified. For a complete list of possible error type/strings, see Table 5-2.
Annotations	Contains a list of annotations entered by the ECPN administrator including the time, administrator name, administrator host, and annotation text

Table 5-17 JDS Viewer Message Journal Pane (Continued)

Field Name	Description
Linkages	Contains a list of message object linkage information such as parent MSN (for system-generated message objects)
GS Info	Contains a list of GS information including the GS From and To addresses and the GS Control Number.
Action Summary	Contains a list of actions taken with the message, along with the time, channel (where applicable), ECPN administrator/host name (where applicable).

The JDS Viewer is not a separate application but is a common library routine available in a C/Motif version. The applications invoking this routine retrieve all message object data through the RPCServer (described in Section 5.1.1) and have the ability to connect to any reachable ECPN RPCServer across a TCP/IP network and obtain message object data stored on that host.

The ECPN administrator has the ability to append annotations to a message object through the JDS Viewer. These annotations include the ECPN administrator name, client host name, time stamp, and text. These append-only annotations are processed through the RPCServer.

5.4.8 Raw Message Viewer

The Raw Message Viewer provides a common GUI for displaying the raw message content (i.e., the content of a message as received). It is invoked from the Message Log Viewer (described in Section 5.4.5), the Error Queue Viewer (described in Section 5.4.6), the channel logs, and the ObjectMMI (described in Section 5.4.14).

For a message received on an X12 channel, the content is extracted from the segments of a message object. For a message received on a UDF channel, the content is extracted from the saved UDF buffer in the message object. In either case, the content is displayed in a single pane with each unprintable character replaced with the octal representation of the byte value.

The Raw Message Viewer is not a separate application but is a common library routine available in a C/Motif version. The applications invoking this routine retrieve all message object data through the RPCServer (described in Section 5.1.1), and have the ability to connect to any reachable ECPN RPCServer across an IP network and obtain message object data stored on that host. For the location of the data stored, see Section 5.4.1.

5.4.9 Channel Log Viewer

The Channel Log Viewer displays the files sent or received on each channel. The following fields are displayed by the Channel Log Viewer:

- Filename
- Byte size
- Time of receipt/transmission

Each field is sortable by clicking the column heading. For the outgoing channel log, the information above is displayed in a file folder format. Opening a folder reveals the list of MSNs and ICNs transmitted in that file. Selection of an MSN invokes the JDS Viewer. Viewing the file content at this level will display a two-pane window, with the top pane listing each MSN transmitted in the file. Within each MSN, the actual GS and ST segments sent from the original ISA envelope are listed. For both incoming and outgoing channel logs, the bottom pane of the window shows the text as it was actually received or sent, to include email headers, MIME encoding, conversions for ISA05/06 overwrite, segment terminator replacement, and single ISA/GS/ST. If it is a UDF channel, the UDF text is shown in the bottom pane. For the location of the data stored, see Section 5.4.3. Both incoming and outgoing logs allow the user to save a file's text to a file on disk, as well as search the file text for a specified string.

The ECPN administrator can select file retransmit to requeue the same GS segments from the original file (described in Section 5.3.6). The content will be placed in the outgoing channel queue for transmission. The retransmit operation can be used to resend messages after changes have been made to the channel's configuration.

Table 5-18 Channel Log Fields (CHNL_LOG_REC)

Field Name	Type	Description
filename	SHORT_FILE_NAME	Filename on remote site (Message ID for email channels)
ftime	long	Time message was received/transmitted from/ to channel
filesize	unsigned long	Size of the file in bytes
data_offset	unsigned long	Offset to data for the message in compressed file
msn_length	unsigned long	Number of bytes of MSN/GS-related data in compressed data file
text_length	unsigned long	Number of bytes containing message in compressed text data file

5.4.10 Email Domain Queue Viewer

This application provides a combined view of the email stats database (see Table 5-7) and the email domain channel queue. The viewer application uses a tree widget with branches representing domains and leaves representing the MSNs in the queue records.

5.4.11 Channel Queue Viewer

The Channel Queue Viewer application provides a listing of records queued to go out a channel queue. The fields displayed in the Channel Queue Viewer are the MSN, the time of queuing, and, for files destined for a UDF channel, the outgoing UDF file name. The raw message viewer (described in Section 5.4.8) or the JDS Viewer (described in Section 5.4.7) may be invoked from the Channel Queue Viewer. The user may delete a queue entry, with the option of placing it into the error queue.

5.4.12 Rejected Email Box Viewer

The RejectedEmail application provides a view of the contents of the rejected email mailbox. The rejected email is stored in the following file: /h/data/global/EC/Messages/rejected. For a description of received email processing and rejection, see Section 5.2.6. The ECPN administrator can select a message to view and can reinject a message after modifying the channel configurations to attribute the message to a particular channel. The Rejected Email Box Viewer displays the From address, the message id, and the reason for email rejection.

5.4.13 RDBMS Injector

The RDBMS injectors, ObjInject and TrnInject, process data that is appended to the RDBMS Queue (OUT_RDBMS_REC) and the Transaction Queue (OUT_TRANS_REC) respectively. These queues are populated by the X12 Message Processing CSC (described in Section 5.3) and the Communications CSC (described in Section 5.2). The Router populates the RDBMS Queue with data associated with the message envelope. The Router, Comms process (described in Section 5.2.2), and EmailSend process (described in Section 5.2.5) populate the Transaction Queue with data associated with transactions performed by ECPN on a particular message (e.g., received, transmitted).

The RDBMS injectors pull data (listed in Table 5-19 and Table 5-20) off their respective queues and inject it into the RDBMS message database (described in Section 5.4.15).

The RDBMS injectors are written in a combination of C and embedded Standard Query Language (SQL).

Table 5-19 RDBMS Queue Table (OUT_RDBMS_REC)

Field Name	Type	Description
msn_name	MSN_NAME	Message Sequence Number (format: SNNNNNNNN/YYYYMMDD)
ic_number	int	Interchange control number
recv_id	ID_NAME	ISA To field
send_id	ID_NAME	ISA From field
gs_ctrl_no	int	GS control number
gs_to	ID_NAME	GS To field
gs_from	ID_NAME	GS From field
st_ctrl_no	ST_NUM	ST control number
trans_type	ST_TYPE	Transaction message type (e.g., 850, 843, 836)
po_number	PO_NUM	Purchase order number
solicitation	SOL_NUM	Solicitation number

Table 5-20 Transaction Queue Table (OUT_TRANS_REC)

Field Name	Type	Description
msn_name	MSN_NAME	Message Sequence Number (format: SNNNNNNNN/YYYYMMDD)
trans_code	int	Type of action (e.g., RECV, XMIT)
trans_dtg	long	Time of action
channel_name	CHAN_NAME	Channel associated with action
byte_count	int	Size of the message
file_name	SHORT_FILE_NAME	Remote side file name

5.4.14 RDBMS Retrieval

The RDBMS retrieval application (ObjectMMI) provides a Motif GUI interface for directing a retrospective search of the RDBMS message database (described in Section 5.4.15). This application constructs a dynamic SQL query (based on data input from the ECPN administrator), executes the query, and presents the retrieved data in a scrolling text window. For any selected entry in the scrolling text window, the user can invoke the Journal Data Summary Viewer (described in Section 5.4.7) or view a list of transactions that have occurred on the message.

The ECPN administrator may enter data values in the following fields to direct the retrospective search:

- MSN
- ISA Control Number
- ISA From
- ISA To
- GS Control Number
- GS From
- GS To
- ST Control Number
- ST Type
- Channel
- Purchase Order Number
- Solicitation Control Number
- Filename
- Date Range

5.4.15 RDBMS Message Database

The RDBMS functions are implemented using an Oracle database. The RDBMS tracks messages based upon message attributes. For example, it is common for site personnel to look up messages based on the Purchase Order Number or Solicitation Number contained within a transaction or by the filename under which the message was transmitted to ECPN. The RDBMS message database contains derived message information on each message processed by the ECPN. The data is stored in two tables: a message object table and a transaction table. The message object table contains ISA, GS, and ST fields from the message object. The transaction table stores information on transactions that occurred for a given message. This database is queried via the RDBMS retrieval application (described in Section 5.4.14). New RDBMS message tables are created each day to hold information on all messages received on that day. The RDBMS table database (see Section 5.4.16) contains information on which RDBMS message tables are currently loaded on the system. The archived tables are stored under Daily/<yyyymmdd>/Archives/ORACLE.

The fields maintained within the message database are listed in Table 5-21.

Table 5-21 RDBMS Message Database Fields

Field Name	Type	Description
msn_count	int	Sequence number portion of the MSN
msn_date	date	Date portion of the MSN
ec_schema_ver	int	Schema version number
isa_cntl_num	int	Interchange Control Number (ISA13)
isa_from	char [ISA_MAX]	ISA Sender ID (ISA06)
isa_to	char [ISA_MAX]	ISA Receiver ID (ISA08)
gs_cntl_num	int	Group Control Number (GS06)
gs_from	char [GS_MAX]	GS Sender ID (GS02)
gs_to	char [GS_MAX]	GS Receiver ID (GS03)
st_cntl_num	char [ST_CNTL_NUM_MAX]	Transaction Set Control Number (ST02)
st_type	char [ST_TYPE_MAX]	Transaction Set ID Code (ST01)
po_number	char [PO_NUM_MAX]	Purchase Order Number (BEG03)
solicit_number	char [SOLIT_NUM_MAX]	Solicitation Control Number (BQT02)
trans_id	int	Unique daily transaction identification
trans_code	int	Indicates receipt, transmission, etc.
msn_count	int	Number of MSNs in the message
msn_date	int	Date of the MSN

Table 5-21 RDBMS Message Database Fields (Continued)

Field Name	Type	Description
trans_dtg	date	Transaction date-time stamp
channel	char [CHANNEL_MAX]	Incoming or outgoing channel
byte_count	int	Number of bytes in message
file_name	char [SHORT_FILENAME_MAX]	Filename in remote system
site_id	char	Site identification character

5.4.16 RDBMS Table Database

The RDBMS table database maintains a list of which RDBMS message databases are currently loaded within ECPN. This information is accessed via the RDBMS retrieval application (described in Section 5.4.14).

The fields maintained within the table database are listed in Table 5-22.

Table 5-22 RDBMS Table Database Fields

Field Name	Type	Description
table_name	char [25]	Name of daily table
activity	char [3]	Action (e.g., created, archived, restored, merged)
act_date	date	Activity date-time stamp
row_count	int	Number of rows in table
file_location	char [1024]	Location of backup file

5.5 Translation

The Translation CSC is responsible for converting incoming (to ECPN) User Defined File (UDF) messages to ANSI X12 messages, as well as converting outgoing (from ECPN) ANSI X12 messages to UDFs. The UDF format represents any format used by a remote system to which ECPN is connecting. The Translation CSC consists of the following SCSCs:

- UDF to X12 Translator (InXlator)
- X12 to UDF Translator (OutXlator)
- Trading Partner Database
- System Setup Database

There are two variations of the translator program running on the ECPN. One variation provides translation from UDFs to X12s for all UDFs sent to ECPN from Government agencies. The other variation provides X12 to UDF translation for X12s to be sent to Government UDFsites from ECPN. For each translation that occurs, ECPN sends positive or negative acknowledgments to one or more email addresses as configured in the edit channel window's ADMIN tab. The trading partner database contains trading partner information necessary for converting UDF addressing schemes to a standard Data Universal Numbering System (DUNS) addressing scheme for X12 messages.

The system setup database holds start and end boundaries for Interchange Control Numbers (ICNs) and Group Control Numbers (GCNs), as well as the ECPN site's Interchange addresses. These items are necessary for generating the X12 envelope during a UDF to X12 translation.

5.5.1 UDF to X12 Translator

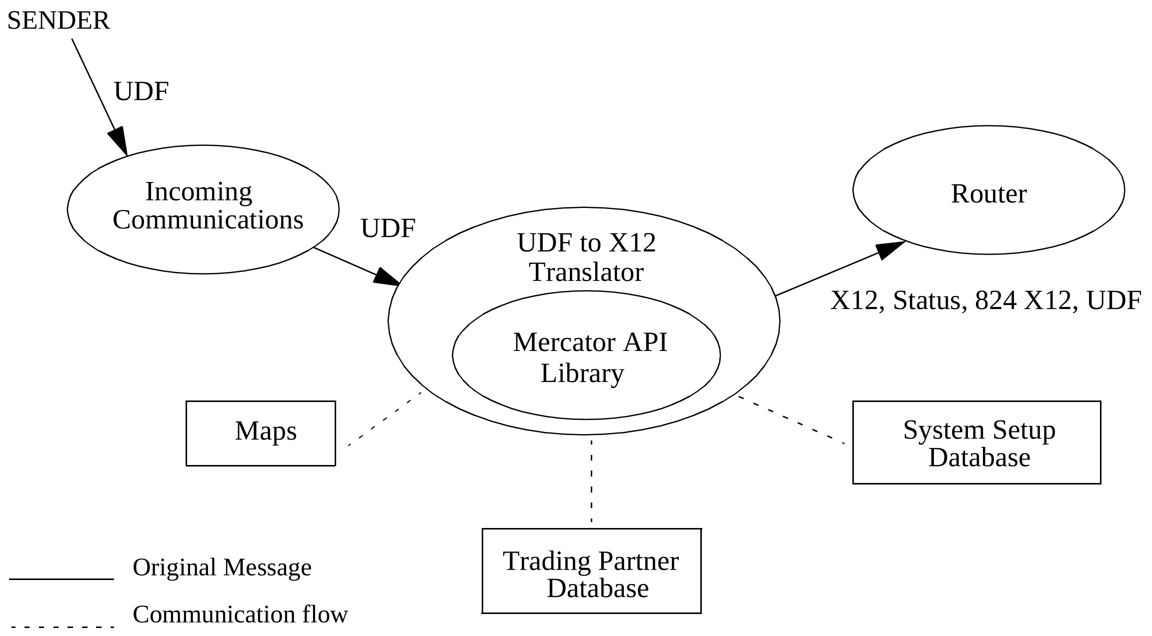
This section describes how a UDF sent to ECPN is translated to an X12 and routed to its destination. The following items are discussed:

- Processing Flow
- Channel Configuration
- Maps and TSI/Mercator
- UDF to X12 Processing Details

5.5.1.1 Processing Flow

Figure 5-2 depicts a UDF being received by ECPN.

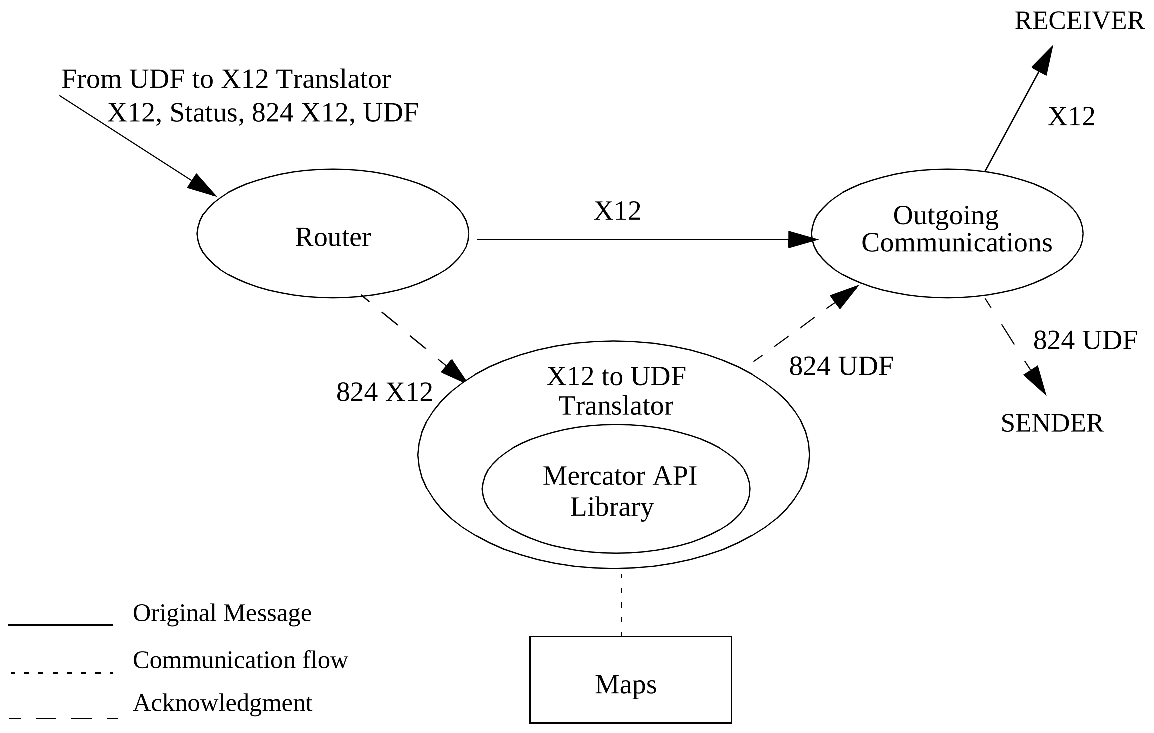
Figure 5-2 UDF to X12 Processing Flow: Translation



The incoming communications processes queue the UDF to the incoming translator for further processing. The translator uses the Mercator API to run the appropriate maps, resulting in the translation of the UDF to an X12. The X12 is passed to the router for processing. See Table 5-24 for the list of items (e.g., 824) placed on the Incoming X12 Queue.

Figure 5-3 depicts the processing done by the router after it receives information from the UDF to X12 translator.

Figure 5-3 UDF to X12 Processing Flow: Decoding/Routing



The router links this information together so that an ECPN administrator can track which X12 was produced from which UDF and what type of acknowledgment was sent to the message originator.

If the translation was successful, the router sends the X12 to the appropriate outgoing communications process to be delivered to the receiver.

If the sender was configured to receive acknowledgments (824 UDF), the router also queues the 824 X12 to the X12 to UDF translator. The translator loads the appropriate maps, translates this 824 X12 ack to a 824 UDF and queues it to the outgoing communications process associated with the sender of the original UDF.

5.5.1.2 Channel Configuration

A communications channel can be configured for receiving/sending only one family of messages (e.g., X12, SPS, SAACONS, APADE). In other words, a channel cannot receive/ send a mixture of messages from different families.

5.5.1.3 Maps and TSI/Mercator

The UDF to X12 translator uses TSI's Mercator API library to:

- bind individual UDFs within a multiple UDF file.
- parse the individual UDFs.
- load the appropriate map, based on the map family configured in the channels database.
- execute the map, resulting in parsing of the UDF and generation of an X12 that conveys the same information as the UDF.

Most of the UDF families have two maps, one for translating UDFs to X12s and another for translating X12s to UDFs. A few families (e.g., those associated with DTS) may have only the X12 to UDF map. Refer to the *Mapper's Guide for Electronic Commerce Processing Node* for details about the mapping process.

5.5.1.4 UDF to X12 Processing Details

1. The communications channel for receiving UDFs from the sender is set to the UDF family that the sender transmits.

2. The associated comms process opens the UDF to X12 translation queue.

3. For each file that is successfully received, during the conversion, the communication process:

   a. creates a queue entry structure of type UDF2X12_QREC, with members as noted in Table 5-23.

   b. appends that queue entry to the UDF to X12 translation queue.

Table 5-23 Comms to Translation Queue Fields for UDF to X12 Translation (UDF2X12_QREC)

Field Name	Type	Description
orig_filename	SHORT_FILE_NAME	Name of the file as it was named on the remote system
udf_filename	SHORT_FILE_NAME	Name of the UDF file on ECPN (a temporary name)
udf_chan_name	CHAN_NAME	Name of the channel on which the UDF file has been received
udf_msg_type	CHAN_MSG_TYPE	UDF family to which the UDF belongs (e.g., SPS)
TOR	long	Time of receipt. ECPN system time when the UDF file was received.
reproc_MSN	MSN_NAME	Message Sequence Number (format: SNNNNNNN/YYYYMDD) used only when this message is being retranslated

Once the session is done, the communication process closes the queue.

4. The UDF to X12 translator process opens two queues: the UDF to X12 translation queue to get entries made by the communication processes and the incoming X12 queue for giving the results to the router.

5. For each entry retrieved from the UDF to X12 queue, the translator:

   a. runs the premap which binds individual UDFs within a multiple UDF file. This step prepares the intermediate UDF files that will be passed through the UDF to X12 map.

   b. forms the X12 file name to which the translated output will be written.

   c. calls the translator support function, “UDF2X12”, to translate the UDF. This function:

      - loads the appropriate map(s) for Mercator, based on the map family configured for the channel.

      - calls functions from the Mercator API library to perform the translation from UDF to X12. During this operation, lookups are performed on the trading partner database and the system setup database in order to generate the X12 envelope.

- generates an 824 “acknowledgment” message, concerning the success or failure of the translation, which is sent back to where the UDF originated.
- interacts with the trading partner database and the system setup database for generating envelope information for the resulting X12.

d. constructs an IN_X12_FILE_REC.IN_UDF2X12_REC entry (see Table 5-24), and places it in the Incoming X12 queue.

6. After all of the individual transactions that comprise the original UDF file have been processed, ECPN deletes the local copy of the original UDF file and removes the processed entry from the UDF to X12 translation queue.

Table 5-24 Translator to Router Queue Fields for UDF to X12 Translation (IN_UDF2X12_REC)

Field Name	Type	Description
genx12_and_824	struct UDF2X12_REC	
x12_filename	SHORT_FILE_NAME	Name of the translated X12 file
udf_filename	SHORT_FILE_NAME	Name of the UDF file on ECPN that resulted in the X12 (individual transaction file)
xltr_error_type	short	Value from the enum XLTR_ERROR_TYPE that indicates the status of running a map
errfile	SHORT_FILE_NAME	Name of the error file that was produced by the translator with detailed errors found while translating the UDF
TOX	unsigned long	Time of translation. ECPN system time when the UDF file was translated
gen824_filename	SHORT_FILE_NAME	Name of the 824 translation status file that is sent back to the sender if so chosen
sectinfo_filename	SHORT_FILE_NAME	Name of a file that has info on “groups” of messages (e.g., for IPC)
pull_filename	SHORT_FILE_NAME	Name of the UDF file on the remote system
in_chan_name	CHAN_NAME	Name of the channel on which this UDF came into ECPN
udf_msg_type	CHAN_MSG_TYPE	UDF family to which this UDF belongs
TOR	unsigned long	Time of receipt. ECPN system time when the UDF file was received.

Translation success, TPDB lookup failure, and UDF errors are appropriately noted in the queue record.

Retranslating a UDF after TPDB errors are corrected:

When TPDB lookup errors occur, 824s are not sent back to the sender. It is the responsibility of the ECPN administrator to correct the trading partner database and retranslate the UDF. Upon retranslation, the router forms a UDF2X12_REXLATE_REC structure, filling in the reproc_MSN field, udf_filename field, and others, and places it in the UDF to X12 translation queue. Because the original UDF file is not kept, the translator pulls the UDF message text from the message object and creates a file for the maps.

The translator then processes this entry just as it would process an entry made by a communication process. The results are sent to the router in the union variant IN_X12_FILE_REC.UDF2X12_REXLATE_REC (see Table 5-10).

5.5.2 Trading Partner Database (TPDB)

ECPN sends and receives messages to and from many different applications. Each application implements a unique UDF. These UDFs may contain unique addressing information that must be converted to standard X12 addressing during translation. Using the UDF address information, the translator performs a lookup on the trading partner database to get the corresponding X12 address.The trading partner database contains a list of addresses for each trading partner that was either parsed from an 838 message or manually entered.
.

Table 5-25 Trading Partner Profile Database Fields for Translation

Field Name	Type	Description
isa_addr	char [ISA_ADDR_LEN +1]	ISA Address
isa_qual	char [ISA_QUAL_LEN +1]	ISA Address Qualifier
gs_addr	char [GS_ADDR_LEN +1]	GS Address
duns	char [DUNS_LEN +1]	Dun and Bradstreet Numbering System code
provider	char [PROVIDER_LEN +1]	Three-letter VAN Identification
cage	char [CAGE_LEN +1]	Commercial and Government Entity Code
dodaac	char [DODAAC_LEN +1]	DoD Activity Address Code

Sites register with the Central Contractor Registry (CCR) with an X12 838 (Trading Partner Profile). The 838 is forwarded to ECPN where it is parsed to create or update trading partner database entries. Table 5-26 lists the fields that are parsed from the 838 for inclusion in the trading partner database.

Table 5-26 CCR 838 Parsed Fields

Segment	Element	Notes
020 BTP	01 -Transaction Set Purpose	00 Original 01 Cancellation 07 Duplicate - Handled as an original
	02 - Reference ID	
	03 - Date	
	04 - Time	
	06 - Transaction Set Purpose (For BTP01 = 00 or 07 only)	04 - Change 30 - Renewal 35 - Request Authority
	07 - Reference ID	
	08 - Date	
	09 - Time	
030 PLA	01 - Action Code	WQ - Accept
060 N1	01 - Name	Must provide one loop with N101 = KK for registrant and N101 = WQ for activity or agency of registrant
	03 - Identification Qualifier	1 - DUNS Number 9 - DUNS+4 Number
390 ENE	01 - Communication Environment Code	PP - Point-to-Point SC - Service Provider (indicates ENE03 identifies a VAN)
	03 - Communication Number	Three-Letter VAN ID
400 N1	01 - Entity Identifier Code	GP - Gateway Provider NN - Network Name
	02 - Name	If N1 = NN, use to indicate full name of VAN
	03 - Identification Code Qualifier	ISA Address Qualifier
	04 - Identification Code	ISA Address

5.5.3 System Setup Database

The system setup database holds the following information that is used by the translator to fill in the ISA05, ISA06, ICN and GCN fields when translating a UDF to an X12:

- Site ID Qualifier (ISA05)
- Site ID (ISA06)
- Starting and Ending Interchange Control Numbers (ICN) (ISA13)
- Starting and Ending Group Control Number (GCN) (GS06)

The ISA05 and ISA06 specified in this database is put into every X12 generated by the translator. For every X12 produced, the translator increments the current ICN and GCN, using the start ICN and start GCN initially and wrapping once the ending ICN or GCN is reached.

5.5.4 X12 to UDF Translator

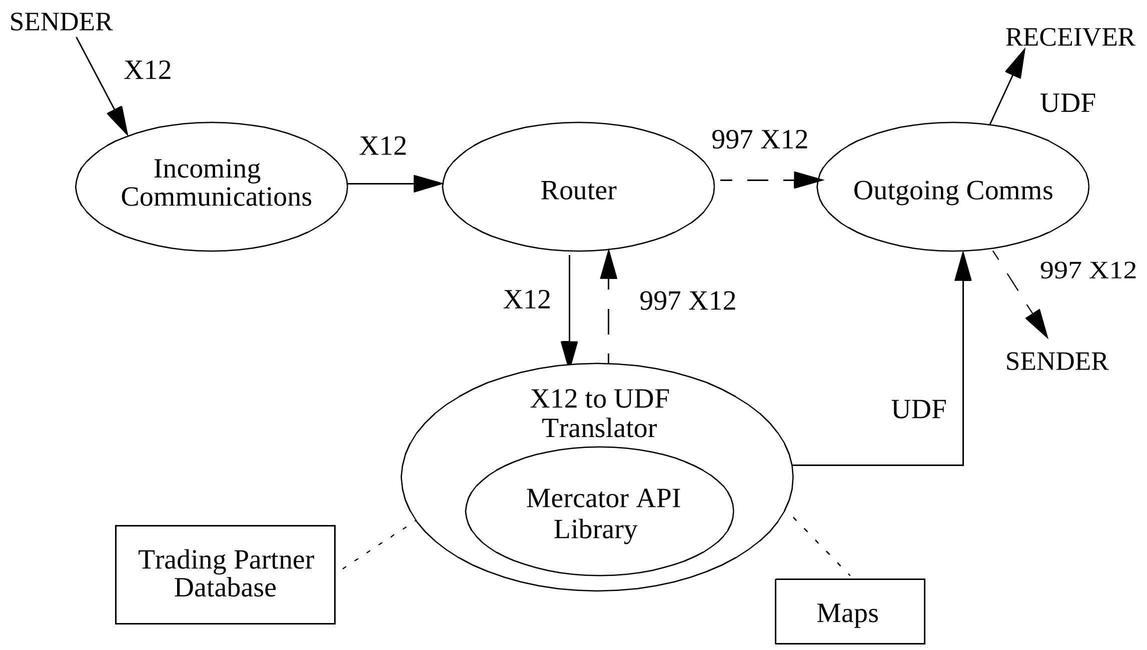
This section describes how an X12 message that is sent to ECPN is translated to a UDF for routing/transmission to a UDF destination. The following items are discussed:

- Processing Flow
- Channel Configuration
- Maps and TSI/Mercator
- X12 to UDF Processing Details

5.5.4.1 Processing Flow

Figure 5-4 depicts an X12 being received by ECPN from a sender. The incoming communication processes queue the X12 file to the router for further processing. After completion of bounding and parsing, the router determines the destinations for the message. For each destination that is configured as UDF, the router queues the X12 to the X12 to UDF translator for further processing. The X12 to UDF translator uses the Mercator API to run the appropriate maps to perform the translation.

Figure 5-4 X12 to UDF Processing Flow: Translation



Upon successful translation, the UDF is sent to the appropriate outgoing communications process for delivery to the receiver. The 997 X12 is sent to the router with its destination being the sender of the translated X12.

If the sender was configured to receive acknowledgments (997 X12s), the router queues the 997 X12 to the appropriate outgoing communications process.

If translation failed, the X12 is place in the error queue for handling by the ECPN administrator.

5.5.4.2 X12 to UDF Processing Details

1. The communications channel for sending the UDFs to the receiver is designated as the UDF family that the receiver can process.

2. The router reads an IN_X12_FILE_REC record from the incoming X12 queue.

3. The router determines the final destination of the message. If the destination channel, as configured in the channel database, is a UDF channel, the router forms an entry of type X122UDF_QREC and appends it to the X12 to UDF translation queue.

Table 5-27 Router to Translator Queue Fields for X12 to UDF Translation (X122UDF_QREC)

Field Name	Type	Description
msn	MSN_NAME	Message Sequence Number that uniquely identifies the X12 message on the ECPN
out_chan_name	CHAN_NAME	Name of the channel on which the UDF file is to be sent
udf_msg_type	CHAN_MSG_TYPE	UDF family to which the X12 should be translated
identifier	int	Used internally to maintain queues

4. The X12 to UDF translator process begins processing the queue entry by using the MSN info in the entry to form a temporary X12 file. The translator support function, “x122udf”, is then called to translate the X12 to a UDF.

Once the translator successfully completes processing the entry, it removes the temporary X12 file it generated and deletes the entry from the X12 to UDF translation queue. The translator constructs an IN_X12_FILE_REC.X122UDF_REC entry for the generated 997 and appends the entry to the incoming X12 queue for the router to process further. The translator also creates and appends an OUT_CHAN_REC entry to the appropriate outgoing channel queues, telling the outgoing communications process to transmit the generated UDF.

On translation failure, the X12 is placed in the error queue, and the entry is removed from the X12 to UDF queue.

Table 5-28 Translator to Router Queue Fields for X12 to UDF Translation (X122UDF_REC)

Field Name	Type	Description
msn	MSN_NAME	Message Sequence Number for the X12 that was just translated
identifier	int	Used internally to maintain queues
gen997_filename	SHORT_FILE_NAME]	Name of the 997 X12 file
xltr_error_type	short	Value from the enum XLTR_ERROR_TYPE that indicates the status of running a map
out_chan_name	CHAN_NAME	Name of the channel on which the UDF was sent

Success or failure is appropriately indicated in the X122UDF_REC.

Table 5-29 Translator to Comms Queue Fields for X12 to UDF Translation (OUT_CHAN_REC)

Field Name	Type	Description
msn_name	MSN_NAME	Message Sequence Number for the X12 that was just translated
out_udf_filename	SHORT_FILE_NAME	Name of the UDF file that was generated from translating the X12
site_id	char [SITE_ID_LEN]	Three-character file extension denoting the site id on the remote and used in naming the UDF file that's pushed to the remote site. (Used for SAACONS only)
xvar	char [XVAR_LEN]	Value passed from the X12 that was translated and used in naming the UDF file that's pushed to the remote site
identifier	int	Used internally to maintain queues
ttype	char [TTYPE_LEN]	(Used by SAACONS only) One-character file prefix that identifies the transaction type and used in naming the UDF file that's pushed to the remote site

5.6 Alert Management

The Alert Management CSC is responsible for providing a single mechanism for generating and managing alerts across the ECPN CSCI. The Alert Management CSC consists of the following SCSCs:

- Alert Daemon
- Alert Notifier (NEPAlertNotify)
- Alert Notifier Database

5.6.1 Alert Daemon

The alert daemon manages alerts generated by the ECPN CSCI. Generated alerts are stored on disk and delivered to clients upon request. Applications request alerts from the alert daemon based on the alert type. The alert daemon is also responsible for marking each alert as having been dismissed or not.

The circular queue used by the alert daemon to manage alerts contains a maximum of 2,500 ALERT_QUEUE_SIZE entries.

Implementation **-** Implemented in the libAlerts library archive and the AlertDaemon server.

5.6.2 Alert Notifier

The alert notifier (NEPAlertNtfy) is a client of the alert daemon (described in Section 5.6.1) that receives alerts and performs the notification actions defined for the alert in the alert notifier database. An NEPAlertNtfy process is started for each user session. The alert notifier performs two notification actions:

- Electronic mail (which optionally includes the data file causing the alert)
- Personal beeper or phone dialing

5.6.3 Alert Notifier Database

The alert notifier database is a data element that defines the notification action(s) to be performed when an alert is generated. An entry in the database consists of the alert criteria and one or more notification actions. The notification methods are via email or dialing a phone number (i.e., notification via beeper or cellular phone).

The alert notifier database is populated through the AlertNtfyDB user interface**.** The alert notifier database is limited to 200 entries, and each entry can contain up to 10 notification actions.

Implementation **-** Implemented in the AlertNtfyDB application.

Table 5-30 Alert Notifier Fields

Field Name	Type	Description
type	int	Type of alert to notify on (corresponds to alert types in NEPAlertList datafile - e.g., "FILE ACCESS ERROR")
key	char[]	Channel to match against before notifying. If set to "ALL", all alerts of the above type are notified, regardless of which channel (if any) they were associated with.
num	int	Number of notification actions set for an alert
method	int	Notification method - EMAIL or BEEPER
active	int	Alert notification activated
address	char[]	Address to email notifications should be sent
phone	char[]	Number to dial for beeper notifications
confirm	int	Flag for whether or not to include file (if present) in email notifications

5.7 Executive

The executive manages all ECPN processes. It launches processes when it receives a request and manages the shutdown of processes when they exit. Requests for process launch come from either a GUI event or ecedi_srv, which is a script that launches the EC_COE services at init level 4.

6.0 Requirements Traceability

The traceability between the ECPN requirements and the version of ECPN in which the requirements were met is addressed in the *Software Requirements Specification (SRS) for Electronic Commerce Processing Node.*

7.0 Acronyms

The following acronyms and abbreviations appear in this document:

AIS: Automated Information System

ANSI: American National Standards Institute

API: Application Programming Interface

ASCII: American Standard Code of Information Interchange

COTS: Commercial Off-the-Shelf

CSC: Computer Software Component

CSCI: Computer Software Configuration Item

CSU: Computer Software Unit

DID: Data Item Description

DISA: Defense Information Systems Agency

DoD: Department of Defense

EBCDIC: Extended Binary Coded Decimal Interchange Code

EC/EDI: Electronic Commerce/Electronic Data Interchange

ECPN: Electronic Commerce Processing Node

Email: Electronic Mail

FIFO: First-In, First-Out

FTP: File Transfer Protocol

GUI: Graphical User Interface

ID: Identification

IDD: Interface Design Description

INRI: Inter-National Research Institute

I / O: Input / Output

JDS: Journal Data Summary

MIME: Multi-purpose Internet Mail Extension

MSN: Message Sequence Number

PCM: Process Control Module

RDBMS: Relational Database Management System

RPC: Remote Procedure Call

SCSC: Sub-Computer Software Component

SDD: Software Design Description

SMTP: Simple Mail Transport Protocol

SQL: Standard Query Language

SRS: Software Requirements Specification

TCP/IP: Transmission Control Protocol/Internet Protocol

UDF: User-Defined File

UTC: Universal Time Coordinate

This page has been intentionally left blank.

Appendix A Alerts

Alerts are generated by ECPN to notify users of specific conditions or problems. A description of each alert is provided in Appendix B of the *Software User's Guide for Electronic Commerce Processing Node.*

Appendix B System Capacities

The following list of system data repositories have unlimited capacities. They are limited only by the amount of disk space available.

- Routing Database
- Channel Database
- Trading Partner Database
- Message Log
- Error Queue
- System Log
- Outgoing Email Queue
- Incoming X12 Queue
- Incoming Translation Queue
- Outgoing Translation Queue
- Interface Status Database
- Message Database

The capacities of other data repositories are as listed in Table B-1.

Table B-1 Capacities

Data Repository	Capacity	Description
Alert Database	2500	Circular queue which wraps at 2500
Alert Notify Database	200 entries with 10 notifications per entry	Maximum number of entries is 200. Each entry can have 10 notification actions.

Appendix C Glossary

Computer Software Configuration Item (CSCI) - A CSCI is a sub-component of a CSC.

Computer Software Configuration (CSC) - A CSC is a sub-component of a System.

Computer Software Unit (CSU) - A CSU is a sub-component of an SCSC

Inbound - Describes messages being received by the government. ECPN is considered part of the government; thus, messages received by ECPN are considered inbound messages.

Multi-purpose Internet Mail Extension (MIME) - MIME extends the format of Internet mail to allow non-US-ASCII textual messages, non-textual messages, multipart message bodies, and non-US-ASCII information in message headers.

Outbound - Describes messages being sent out by the government. ECPN is considered part of the government; thus, message sent out by ECPN are considered outbound messages.

Remote Procedure Calls (RPC) - RPCs provide a way to distribute program segments across computers in a network. This allows communication with more than one machine on a given network while executing a program and communicating with other programs that run on the same machine.

Simple Mail Transport Protocol (SMTP) - SMTP is Internet’s standard host-to-host mail transport protocol.

Sub-Computer Software Component (SCSC) - A SCSC is a sub-component of a CSCI

User Defined File (UDF) - User defined file is a generic term that applies to any Electronic Commerce system that does not output data according to a defined EDI format (e.g. X12 or EDIFACT). By definition, UDFs are different for different systems.

Appendix D Message Object Parse API

The following API(s) are provided to simplify parsing of the message object segments:

int AddKey(char *seg, ...)

“seg” represents the segment type of interest to the caller. “...” is a variable length list of the field numbers in that segment that the caller wants returned. The last item in this list must be a -1 as a sentinel value for the list. A field number of 1 identifies the field after seg. AddKey() returns 1 if successful in adding the key, 0 otherwise. Possible causes of failure:

The key list is full. Currently, the caller can specify a maximum of 10 keys.

seg has a string length greater that 4, the number of bytes in an integer.

Example: AddKey(“GS”, 5, 8, -1);

This example instructs that, for every GS in the message object, return fields 5 and 8 to the caller.

void ClearKeys()
Used to clear all the current keys added by the caller.

int SetMSN(char *msn_name)
Used to set the message object, specified by “msn_name”, on which to perform the key search. Returns 1 if successful, 0 otherwise. Possible causes of failure:

Could not open the message object specified.

unsigned int FindSeg()
Used to find the next segment which matches any of the keys added by the caller. Returns the SegVal of the segment matched, which is equivalent to HASH(“<seg str>”). Returns 0 if unsuccessful. Possible causes of failure:

Reached the end of the message object.

void SetMsgObj (EC_MSG_OBJ *m)
Identifies a new message object to parse. Resets all current state information, except the keys. Parse routines will now start at the beginning of the message object, looking for previously set keys.

void FreeFields (char **ptrs, int num)
Frees the list of fields which were allocated by a call to FetchFields(). “ptrs” is the list of fields and “num” represents the number of fields in the list.

int FetchFields(char **fields)
Used to return the fields requested by the caller for the currently matched segment. The number of char *field pointers should equal the number of fields requested by the user in the AddKey() call, and the return order of the fields is the order in which the fields were listed in the AddKey() call. Any field that was unable to be retrieved will be returned as NULL. Returns the number of fields successfully retrieved for the segment.

NOTE: HASH exists both as a macro (typical usage) and as a C function (for use in debuggers). UNHASH only exists as a C function.

unsigned int HASH(char *str)
HASH is a macro function which will return an unsigned integer representing the content of "str".

char *UNHASH(unsigned int IdVal)
Returns the string value for "IdVal".

The following is a sample code segment representing the usage of these functions in a context:

```
#include <string.h>
#include <strings.h>
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <varargs.h>
#include <EC/X12Msg.h>
#include <EC/msg_obj.h>
#include <EC/Vids.h>
#include <assert.h>
#include "parse.h"
#include <EC/common.h>


void main(int argc,  char**argv) {

    char    msn[18];
    char    *isa_fields[3], *iea_fields[2], *ref_fields[2];

    int             i, no_isa = 3, no_iea = 2, no_ref = 2, no_fetched = 0;
    unsigned int seg_type;

    if (argc < 2) {
        printf("Usage: parse <full msn path/filename>\n");
    }

    ncpy (msn, argv[1], 17);
```

```
AddKey("ISA", 1, 5, 15, -1);
AddKey("REF", 1, 2, -1);
AddKey("IEA", 1, 2, -1);

if (!SetMSN(msn)) {
        printf("Couldn't open msn %s.\n", msn);
        exit(0);
}

while ((seg_type = FindSeg()) != 0) {
        if (seg_type == HASH("ISA")) {
                no_fetched = FetchFields(isa_fields);
                if (no_fetched != no_isa) {
                        printf("Only fetched %d of %d fields requested.\n",
                                no_fetched, no_isa);
                }
                for (i = 0; i < no_isa; i++) {
                        if (isa_fields[i] != NULL) {
                                printf ("ISA - %s\n",isa_fields[i]);
                        }
                }
                /*
                ** Because FetchField() strdups fields found, they must be
                ** freed.
                */
                FreeFields(isa_fields, no_isa);
        } else if (seg_type == HASH("IEA")) {
                no_fetched = FetchFields(iea_fields);
                if (no_fetched != no_iea) {
                        printf("Only fetched %d of %d fields requested.\n",
                                no_fetched, no_iea);
                }
                for (i = 0; i < no_iea; i++) {
                        if (iea_fields[i] != NULL) {
                                printf ("IEA - %s\n",iea_fields[i]);
                        }
                }
                /*
                ** Because FetchField() strdups fields found, they must be
                ** freed.
                */
                FreeFields(iea_fields, no_iea);
        } else if (seg_type == HASH("REF")) {
                no_fetched = FetchFields(ref_fields);
                if (no_fetched != no_ref) {
                        printf("Only fetched %d of %d fields requested.\n",
                                no_fetched, no_ref);
```

```
                    }
                    for (i = 0; i < no_ref; i++) {
                            if (ref_fields[i] != NULL) {
                                    printf ("REF - %s\n",ref_fields[i]);
                            }
                    }
                    /*
                    ** Because FetchField() strdups fields found, they must be
                    ** freed.
                    */
                    FreeFields(ref_fields, no_ref);
            } else {
                    printf("Unrecognized segtype\n");
                    break;
            }
    }
}
```

This page has been intentionally left blank.